TEC-0021

US Army Corps of Engineers
Topographic Engineering Center

An Image Understanding Environment for DARPA Supported Research and Applications, Second Annual Report

Douglas Morgan
Advanced Decision Systems
1500 Plymouth Street
Mountain View, CA 94043-1230

Daryl Lawton
Georgia Institute of Technology
225 North Avenue, NW
Atlanta, GA 30332-0320

May 1992

Approved for public release; distribution is unlimited.

Prepared for:
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209-2308

Monitored by:
U.S. Army Corps of Engineers
Topographic Engineering Center
Fort Belvoir, Virginia 22060-5546

93

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE May 1992	3. REPORT TYPE AND DATES COVERED Technical Report Sep. 1990 - Sep. 1991

4. TITLE AND SUBTITLE

An Image Understanding Environment for DARPA Supported Research and Applications, Second Annual Report

5. FUNDING NUMBERS

DACA76-89-C-0023

6. AUTHOR(S)

Douglas Morgan[1] Daryl Lawton[2]

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

[1]Advanced Decision Systems, 1500 Plymouth Street, Mountain View, CA 94043-1230

[2]Georgia Institute of Technology, Atlanta, GA 30332-0320

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Blvd., Arlington VA 22209-1155
U.S. Army Topographic Engineering Center
Fort Belvoir, VA 22060-5546

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

TEC-0021

11. SUPPLEMENTARY NOTES

Effective 1 October 1991, the U.S. Army Engineer Topographic Laboratories (ETL) became the U.S. Army Topographic Engineering Center (TEC).

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report on Image Understanding Environment (IUE) describes the design of the base classes, the structure of the class hierarchy and the user interface. The design benefits the IUE by allowing integration of the diverse concepts of IU within one environment, rapid introduction of new users to the IUE, and an organized extension of the base IUE developers (including new users).

14. SUBJECT TERMS

Vision environment, Image Understanding (IU), Computer Vision (CV), object oriented programming, C++

15. NUMBER OF PAGES

94

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This research is sponsored by the Defense Advanced Research Projects Agency (DARPA), 1400 Wilson Boulevard, Arlington, Virginia 22209-2308 and monitored by the U.S. Army Topographic Engineering Center (TEC), Fort Belvoir, Virginia 22060-5546, under Contract DACA76-89-C-0023, by the Advanced Decision Systems, 1500 Plymouth Street, Mountain View, California 94043-1230, and Georgia Institute of Technology, 225 North Avenue, NW, Atlanta, Georgia 30332-0320. The Contracting Officer's Representative was Mr. George Lukes. The DARPA Program Managers were Dr. Rand Waltzman and Mr. Oscar Firschein.

Section 1

INTRODUCTION

This is the annual technical report on work performed by the Advanced Decision Systems (ADS) Division of Booz-Allen & Hamilton (BAH) on the "An Image Understanding Environment for DARPA Supported Research and Applications" project DACA76-89-C-0023 during the period of September 26, 1990 to September 25, 1991.

The focus of activity during this period has been helping to design the DARPA sponsored Image Understanding Environment (IUE). ADS and both its subcontractors, Georgia Institute of Technology (GT) and Stanford University, are participating in this DARPA Image Understanding (IU) community design effort. The principal design participants have been Mr. Douglas Morgan (ADS), Dr. Tod Levitt (ADS, through September 1991), Dr. Daryl Lawton (GT), and Dr. Thomas Binford (Stanford).

The IUE will significantly advance the state of the art in environments and development frameworks. Central to this advancement is the capability to seamlessly integrate the numerous concepts of IU into one system, to easily introduce new users to the system, and to extend the system in multiple new directions. Our work has focused on providing these capabilities with a class hierarchy clearly embodying IU concepts and a user interface allowing complex interactions to be simply expressed.

During previous reporting periods, the ADS/GT team designed a Vision Environment system and developed a prototype for a Sun workstation in C++. Plans were made for significant improvements to the ADS Vision Environment system prototype. However, the current reporting period began with a large cut in contract funding, leaving only enough support for scaled back design activities. At that time it was clear that further design activities would achieve the greatest impact if oriented toward influencing the emerging DARPA IUE design. Transitioning the Vision Environments design to the IUE and then continuing to refine the design would lead to the tangible result of a better IUE system end product. Like the Vision Environment system, the DARPA IUE is intended to facilitate the transfer of technology from the DARPA IU community into industrial, military, and commercial applications.

The IUE design is primarily being designed by a group of ten DARPA IU community representatives from industry and academia. The design group consists of:

Joe Mundy (Chair)	- GE
Thomas Binford	- Stanford
Terry Boult	- Columbia
Al Hanson	- U Mass
Bob Haralick	- U of Washington
Charlie Kohl	- AAI
Daryl Lawton	- Georgia Tech
Douglas Morgan	- ADS
Keith Price	- USC
Tom Strat	- SRI.

The design efforts by ADS, GT, and Stanford representatives have been funded largely through this contract. Prior to the formation of the design group, several meetings (open to the DARPA IU community) were held to refine the IUE goals and development strategy. During this period, Tod Levitt was instrumental in transitioning the Vision Environments design and design concepts into the IUE and in shaping the direction of the IUE design to initially focus on specification of a comprehensive class hierarchy. Presentations were also given by Tod Levitt and Daryl Lawton at the 1990 IU Workshop. One presentation reviewed the Vision Environments design and prototype development. The other described the functionality that a modern IUE should provide. Rand Waltzman and Oscar Firschein have been the DARPA program managers for the IUE effort.

This document describes the inputs of ADS team to the IUE design. These inputs were primarily in the definition of portions of a class hierarchy for the IUE. ADS was primarily responsible for Object Abstraction (the root classes of the hierarchy). GT was responsible for User Interfaces. Stanford provided support for both areas and additional inputs for Image Features. ADS also has acted as a liaison between the IUE design committee and the DARPA Open Object-Oriented Database project being carried out by Texas Instruments. All three organizations, with GT taking the lead, have helped refine the initial concepts for general spatial objects.

This document contains four sections. Section 2 provides background for the design goals and choices of the last two sections. Section 3 describes the Object Abstraction inputs to the IUE design. Section 4 describes the User Interface inputs.

Section 2

Background and Design Principles

The inputs of the ADS team to the IUE design are based on extensive experience in applying object-oriented techniques to IU. This chapter briefly describes some of the systems with which we gained this experience and presents several high-level design principles we have found to be critical to success.

The ADS inputs to the IUE design are based on experience in developing several generations of Object-Oriented class hierarchies and systems for Image Understanding (IU). These systems include:

- **Vision Environments.** This system was designed and prototyped during the first year of this contract. It was aimed at achieving high portability and cost effectiveness through use of C++ and off-the-shelf class libraries and tools. The design contained a early versions of spatial objects. The prototype integrated the Interviews class library with image handling, image display, and relational database access.
- **Sensor Algorithm Research Expert System (SARES) Testbed—CLOS Version.** This system has been under development with Wright Laboratories and DARPA since 1986. It integrates 3D objects, imagery, feature extraction, interpretation networks, Bayesian Networks, stochastic coordinate transformations, a multiple hypothesis (multiple worlds) mechanism, rule-based control, user interface (with image display, rendering, graphic overlay, network display, tables, and plots). This system addresses requirements, spanning: IU, environments, physical modeling, sensors, estimation, detection, classification, inference, user interfaces, and information management. Beginning about two years ago, ADS began a major reorganization of the Testbed to unify the approach of providing this wide range of capabilities. Since then, we have made major revisions to the testbed to make it reflect a more integrated view. The experience gained here has been a significant factor in the ADS inputs to the IUE design.

- **Sensor Algorithm Research Expert System (SARES) Testbed— C++ Version.** An effort is currently underway to convert a portion of the CLOS SARES Testbed to C++ .

- **View.** View, initially created in 1988, is an extensible class hierarchy (with efficient iteration macro environments, or "IU constructs") for image and image feature objects. It is an outgrowth of Powervision (see below), made with the goals of porting Powervision from ZetaLisp to CommonLISP and of separating the IU processing aspects of Powervision from the interactive user environment aspects. It supplies constructs for efficient iteration over all its spatial objects (images, edges, edges-lists, regions, etc.). Having been used on more than ten ADS projects, including SARES, View has influenced our contributions to image designs and efficiency considerations.

- **Powervision.** Developed and extended by Daryl Lawton while at ADS, Powervision is one of the first object-oriented IU environments. It is written in ZetaLisp/Flavors for the Symbolics Lisp Machine. A distinctive feature of this system is the extensive use of databases for library functions, processing results, and processing history.

- **ADS IUlab.** This is a system of abstract data types (ADTs) in C for types such as Image, Region, ConnectedComponent, ContainmentTree, and Table. This 1985 system implements strict ADTs in C, documenting the few nonfunctional (side-effect or procedural) access paths to object internals. It supports all C primitive numerical types (e.g. unsigned short and double) with typing macros. It supports different boundary handling techniques: reflection, constant value, and border value. To address memory constraints of early VAX systems, an Image can be read/created in any of three modes for disk buffering: all in-memory, read rows from disk on demand (with iterators making transparent access to disk), and read rows from disk on demand with back-up to a temporary disk file. This system also influenced our contributions on efficiency considerations.

With each system experience was gained in development, performance, training, portability and maintenance. Our proposals for the IUE design have been aimed at using the design principles distilled from these developments to arrive at a widely useful IUE. The issues in class hierarchy design that we considered include:

- **Aggregating names for similar concepts.** To ease training and maintenance, a system should consistently name implementations of similar concepts. An example from CommonLISP where naming is not consistent is the set of access functions for the various attribute/value data structures. These access functions include: aref, assoc, elt, get, getf, gethash, nth, nthcdr, rassoc, and slot-value. Although these all do the same conceptual operation (evaluate an index/value association), there are functions with ten different names and the order of arguments changes unpredictably between functions. Without conscious steps to aggregate names, the IUE could easily generate many dozens of more names for the same operation. A tightly knit class hierarchy is an excellent way to encourage consistent naming in the IUE.

- **Specifying precise semantics.** To make it possible to do up-front design and to write comprehensive test suites, the class documentation needs to precisely define the over-time and between-object behaviors of objects. It is especially important to document what to expect for identity, creation, destruction, copying, assignment, equality, persistence, and mutation of objects. Consistency of these aspects of object behavior will greatly affect the ability to integrate contributions from a distributed set of developers.

- **Taking a set- and relation-theoretic approach.** We find that sets, relations (tuple sets), functions (functional relations), and networks (sets of relations) are fundamental to many aspects of the SARES Testbed and our other IU environments. These aspects include:

  - Databases (set of relations)
  - Bayesian Networks (set of conditional probability relations)
  - User interface (sets of object-display-action relations)
  - Coordinate system networks (sets of coordinate transform functions)
  - 3D models (sets of attachment relations)
  - 3D primitives (sets of points)

  By capturing this commonality in a single set-theoretic class hierarchy, names for concepts are tightly aggregated and much of the system's semantics can be specified for just one class and reused many times without change. A single narrow-rooted class hierarchy also allows the objects of a strongly typed language (such as C++) to participate in a wide range of activities defined along the trunk of abstract set-theoretic classes.

Although we had earlier recommended that C++ be the sole delivery language for the Vision Environment system, for the IUE a mix of CommonLISP and C++ is a requirement. CommonLISP adds the significant advantages of automatic garbage collection, dynamic compilation, dynamic loading, interpreted operation, and extensive support for multiple inheritance. These factors, in addition to the reliance of many DARPA IU contractors on CommonLISP, will help with timely construction of robust software for complex IU applications.

Section 3

Object Abstraction

This chapter presents the Object Abstraction inputs to the IUE design. These inputs have been incorporated into the IUE design document produced by the IUE design committee and into the IUE Overview paper presented at the 1992 DARPA IU Workshop.

3.1 Overview

Object Abstraction for the IUE lays out the trunk of the class hierarchy and proposes *software development guidelines*. Its aim is to smooth the way for independently developed IUE components to work together. The IUE class hierarchy is organized around a single root class (*Object*), a metaclass (*Class*), and a core set of classical mathematics, physics, and information processing related classes. Guidelines for extending the class hierarchy are also included.

The IUE will be a large and extensible system jointly developed at numerous sites throughout the country. Bringing order to the IUE build will require one specific class hierarchy and one specific set of design principles. This will provide coordination well beyond the generic practice of "object-oriented programming" or "OO design". The hierarchy and design principles of the IUE must support the built-in IUE capabilities and accommodate new additions. A good foundation will keep the overall costs of building the IUE down and will improve chances for a successful, robust system.

Object Abstraction aims to help developers provide consistent capabilities and names for classes and methods. It also aims to ensure that capabilities are comprehensive and logically arranged with well-defined paths to obtaining maximal efficiency. The overall effect is to allow each new development to be added to the class hierarchy at the logically correct point (rather than, for example, making a new incompatible class hierarchy or work with none at all).

Object Abstraction is essentially one *Object* class, one *Class* class, a *Collection* class hierarchy, and design/programming guidelines for the IUE build. These foundation

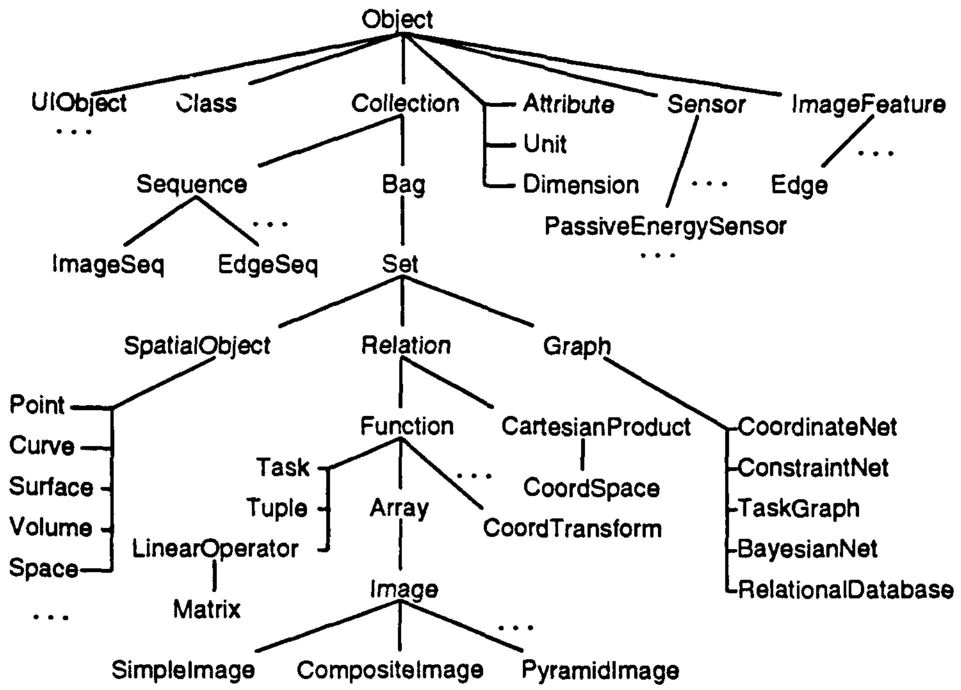


Figure 3-1: Object Hierarchy Outline.

classes define an extensive set of methods for interacting with environment-level tools, for tailoring specialized classes via parametric hooks, and for consistently accessing common mathematics and physics operations. Figure 3.1 outlines the class hierarchy. The figure is an outline of abstract classes: the IUE will have more classes to provide various implementations of the classes shown in the figure. Also, aspects such as name choice, specific inheritance paths, and the use of multiple inheritance are being further refined.

3.1.1 *Object* and *Class*

Together, *Object* and *Class* define the environment-level behavior shared by all objects. All IUE classes inherit from *Object* and are associated with a unique instance of the class *Class*. These classes allow the environment to examine instances and configure operations (especially I/O, copy, and display/editing) based on different types and classes. This two-class approach is closely aligned with systems such as CLOS, Smalltalk, and the National Institute of Health C++ Class Library. Applying the approach to C++ requires substantial infrastructure and discipline, but will be valuable in organizing the IUE and providing powerful interactive capabilities.

3.1.2 Math, Physics, and Information Processing Classes

The math, physics, and information processing classes represent such fundamental concepts as images, extracted features, world objects, pure geometry, transformations (including coordinate systems), sensors, sets, sequences, relations, and networks of relations. The *Collection* class is basic to much of the class hierarchy.

Collection is parameterized with functions (including equivalence, insert compatibility, and union compatibility) so that it can cleanly specialize in multiple directions:

- Finite, countably infinite, and uncountable (at least conceptually so) numbers of elements
- *Object*-valued or language-primitive-valued (e.g., int) element types
- Constraints on types or values of elements to be inserted.

Early versions of the IUE design had nearly every class branching of the trunk on Figure 3.1 inheriting directly from *Object.* The current designs enforce much greater uniformity among important mathematics and physics classes by moving them to more meaningful positions further down the class hierarchy.

3.1.3 Development Guidelines

The development guidelines of Object Abstraction include:

- A dictionary of translating between terminology for C++ , CommonLisp, and standard OOP.
- Naming conventions for methods — these specify such characteristics as return type, inlining, image boundary handling, immutable versions, etc.
- Notions of abstract type and implementation hierarchies embedded in the class hierarchy — this separates method definitions from slot definitions so that highly constrained objects far down in an inheritance hierarchy can have the union of all supertype methods names without carrying around extra slots defined in many unrelated implementations of supertypes.
- Use of source preprocessing tools to aid C++ development.
- Use of persistent object techniques.
- Views of objects — objects wrap around other instances to change the apparent set of methods or interface (e.g., a view is a low-cost and consistent way of creating a vector-valued image from a sequence of images and visa versa).

The guidelines aim at producing efficient class hierarchies free of semantic conflicts on methods or slots (i.e., potential conflicts due to multiple inheritance are not left chance).

3.2 Classes and Conventions

This section shows how the IUE class hierarchy is organized around a single root class (*Object*), a single metaclass (*Class*), and a variety of math and physics oriented classes. *Object* and *Class* work together to define the environment-level behavior shared by all objects. The math and physics classes define the fundamentals of world objects, relations, sensing, networks, and image understanding. Classes for I/O channels and display/interaction devices are not covered.

To help with the organization of a powerful and uniform environment, we propose effective approaches for several areas, including:

- Vocabulary for the object system (especially since we use both C++ and CLOS in the IUE)
- Capabilities that all objects share
- Tools to mechanically generate code required by the environment (such as for class object creation, instance copying, and I/O)
- Naming conventions (especially with regard to how names map into combinations of auxiliary concepts such as inlining, declaring the return type, dynamic binding, static binding and function name overloading, mutable versus immutable objects, free versus member functions)
- Hierarchy that introduces math and physics related methods early on in a few central classes (e.g., `hasMember` for a collection can be used many levels down in the hierarchy to test whether a point is inside 3D object). The hierarchy defines the primary interfaces for each object type (declaring, for example, that an *Image* has primarily a functional interface rather than that of a kind of 3D object with pose.)
- Parametric freedoms allowing the *Collection* class to be specialized to such diverse objects as sequences of arbitrary *Objects*, `unsigned char` valued arrays, and procedurally defined *Functions*
- Network classes sufficient for managing constraints, relational databases, and Bayesian inference networks
- Abstract classes for separating method interfaces defined for a class from the internal representations used by instances. This is needed so that highly constrained objects far down in an inheritance hierarchy do not have to contain the union of slots for all the complex, less constrained objects above.

- View objects for resolving method name conflicts that arise when two logical views of one instance would naturally use the same method name, but for different purposes.

Each of the above issues is now addressed.

3.2.1 Vocabulary

Table 3-1 lists C++ and CLOS names for approximately equivalent concepts. The table also lists our preferred usage in the left column.

3.2.2 Common Capabilities

The capabilities common to all classes are organized around a single root superclass and a metaclass (a class whose instances each describe a unique class). The approach using these two classes is very close to that followed by CLOS, the C++ National Institute of Health Class Library (NIHCL) and Smalltalk. Applying the approach to C++ requires considerable discipline. For CLOS, it is relatively easy.

We base all IUE classes on a single root superclass, *Object*. There are at least two strong arguments doing so. First, in OOP, this is a natural way to be sure that every instance has at least the minimum interface to interact with the standard tools of the environment. Second, without a root *Object* class, the strong typing of C++, forces the use of parameterized versions of subclasses of *Collection* to store instances from each disjoint inheritance hierarchy. Storing instances from disjoint inheritance networks in one collection requires type-violating casts.

We have also chosen to define one class (a metaclass) whose instances represent the characteristics of some other class. An environment often needs to examine the structure of an object and choose operations based on different types and classes. As C++ provides no default way to query an instance about its class, we define one.

3.2.3 Code Generation

It is critical to make creating a new class easy. In C++, this almost certainly implies tools to scan *.h files and mechanically generate code for such activities as creating a class object, and defining the virtual functions supporting instance copying, I/O, display, editing, etc. Although this section points out the need for such tools, it does not present their design.

Table 3-1: Vocabulary Choices for Object-Oriented Programming Concepts.

Preferred Names	Other Names and Usages
Virtual Function	Generic Function (with Dispatch on Only First Argument), Message, Virtual Member Function
Generic Function	Multimethod, Would be a Virtual Function on more than One Argument (This Restricted Notion of Generic Function is Nonstandard, but Makes Clear a Useful Distinction.)
Instance	Object
Class	No Other Name
Method	Virtual Function Definition
Call a Virtual Function	Send a message
Call a Generic Function	Method Dispatch
Self	First Argument of a Generic Function
Method Combination	Daemons (no built-in C++ analog)
Superclass	(Virtual) Base Class (C++ and CLOS Handle Conflicting or Ambiguous References to Slot and Methods Differently)
Direct Superclass	Direct Virtual Base Class
Indirect Superclass	Indirect Virtual Base Class
Subclass	Derived Class
Direct Subclass	Direct Derived Class
Indirect Subclass	Indirect Derived Class
Nonvirtual Base Class	No good CLOS Analog
Abstract Class	Mixin (Eiffel uses Deferred Class)
Nonvirtual Member Function	Statically Bound Member Function (CLOS does not directly support static overloading of function names. Also, CLOS on specializes on a fixed number of arguments for all methods)
Slot	Member Object, Instance-Allocated Slot (bInstance Variable
Static Member Object	Class-Allocated Slot (Smalltalk uses Class Variable)
Free Function	No Other Name
const Object	Immutable (no associated CLOS syntax)
inline Function	Macro Expanded or Inlined

3.2.4 Naming Conventions

We use several naming conventions in this section. The conventions (adapted from X-Windows standards) are:

- The main part of class names begin with a capital letter. Class names with multiple words are concatenated with each word capitalized. Sometimes modifiers, such as component datatype names, are appended to the main part after an underscore. An example class name is *SomeKindOfImage_int.*
- Slot names are all lower case and, if multiple words long, have words separated by single underscores. An example slot name is *this_is_a_slot.*
- Function names have the following format (bracketed items are optional):

  mainPart[*_ReturnType*] [*_PostfixModifier*]

- The return type of a function may be specified by name after the main part of a function name and before the postfix modifiers.
- Postfix modifiers of function names are (at least) of the form:

  `[_[[n]c][[n]i][[n]m][[n]v]]`

  where:

  - `n` - Not
  - `c` - Const (function does not modify first argument's value)
  - `i` - Inline
  - `m` - Member Function
  - `v` - Virtual

- The characters <T> can be freely replaced with the name (possibly using an encoding scheme) of any standard C++ or Lisp datatype (e.g., **unsigned_char** or **double**).

The postfix modifiers allow the developer to consistently name the multiple different functions (up to 16 for one *mainPart* name) that perform one logical task, but satisfy different requirements for speed, safety, time of binding, and ability (in C++) to reference functions by pointer. (Actually, the "member" and "not member" options are redundant since C++ allows both types of functions to be named identically and still be uniquely referenced using "::" syntax.) Other modifiers options can be added to indicate choices such as image access boundary handling (e.g., no checking, reflection, wrapping, default value).

The postfix modifiers have no defaults. That is, the exact meaning of an unadorned function varies from function to function. The designer is free to choose the unadorned name for the most commonly used version of the function. The `n` option can negate any of the `c`, `i`, `m`, or `v` options implicitly chosen for the unadorned function.

In C++ and CLOS, the return type of a function does not enter into the generic or virtual function method dispatch calculation. For efficiency, we often want to have logically similar functions be implemented completely differently depending upon desired output types. E.g., an inline method that returns a `char` from a `char` array object could be many orders of magnitude faster than a similar method that returns an instance whose state represents a `char`. The naming conventions address this need with the *ReturnType* portion of the function name.

The methods chosen to be defined in this section do not use postfix modifiers. Modifiers will be important to the final IUE, but are not essential to the presentation of an overall logical design.

3.2.5 Hierarchy

The class hierarchy (of Figure 3.1) includes the root *Object* class, the *Class* class, and math and physics related classes. The figure shows the hierarchy only to depths where it touches the top level classes being designed by other organizations represented on the IUE design committee. The hierarchy shows that an *Image* inherits from *Array*, *Function*, *Relation*, *Set*, *Bag*, *Collection*, and *Object*. This structure prescribes a host methods for *Image*. The definitions of many methods will be inherited, the remainder are supplied by the implementations of *Image* and its subclasses.

3.2.6 *Collection* Class

The *Collection* class is basic to much of the hierarchy. It must cleanly specialize in multiple directions. Different subclass need to represent

- Finite, countably infinite, and uncountable (at least conceptually so) numbers of elements
- *Object*-valued or language-primitive-valued (e.g., int) element types
- Constraints on types or values of elements to be inserted.

Parameterization by two functions makes *Collection* sufficiently flexible. These functions are the equivalence function and insert compatibility test. Additional useful parameterizations are a union compatibility test (actually derivable from the insert compatibility test) and a potential constraint on element type.

3.2.7 Network Classes

Relational databases, Bayesian networks, function composition networks, and constraint networks are all networks for managing joint information over a set of attributes. The structures necessary to properly support these networks extend well beyond the usual undirected and directed graphs of links and arcs connecting two nodes at a time. (We choose to use "link" rather than "edge," the standard, to avoid clashing with the image related usage of "edge.") Of particular importance are generalizations of links and arcs to connections between sets of nodes. These are the hyperlink, or set of nodes, and what we will call the hyperarc, or directed pair of sets of nodes. A set of hyperlinks forms a hypergraph and a set of hyperarcs forms what we will call a dihypergraph (directed hypergraph). Also necessary are attributes (nodes) with their domain sets. Further, a structure at the heart of efficient inference and query algorithms is the Join tree of hyperlinks (undirected tree of hyperlinks with the intersection of any two hyperlinks contained in every hyperlink in the connecting path).

3.2.8 Abstract Classes

In this description of the hierarchy, *Collection* and its subclasses are abstract classes, that is, classes that define method name and signatures, but do not provide slots and implementation. This allows objects of widely varying implementation but identical interface to be created with no superfluous per-instance overhead. As the IUE develops, the hierarchy will be filled in with multiple subclasses supplying specific implementations for each abstract interface.

3.2.9 Views

With a complex hierarchy built largely on sets, relations, and functions, it is inevitable that one logical view of some class will involve a set, relation, or function interface, but not in the way implied by the inheritance hierarchy. For example, the inheritance defined view of a *BayesNet* is as a directed hypergraph with set-like interface to a structured set of *Hyperarcs*. Because a *BayesNet* can logically be viewed as a joint probability density (a function), it could also have a set-like interface to a structured set of *Tuples* (mapping random variable values to real numbers). A *View* object would wrap around the *BayesNet* instance and convert the *Set* interface methods of union, insert, etc. to operate on the set of *Tuples* instead of the set of *Hyperarcs*. We propose that view creating methods (such as *asFunction*) be provided as needed to wrap instances with new instance with modified interfaces. The view objects of Interviews are similar to our *View* objects, but they are used only to redefine the user interface methods of an object instead of any set of methods. This section will not specify view objects other than to say that there will be many view subclasses and their use should be an integral part of the environment.

3.3 Hierarchy Class Definitions

3.3.1 Object

The *Object* class is the root superclass of all classes in the environment. In conjunction with the *Class* metaobjects, *Object* specifies the interface for the environment-level operations that apply across all instances. These operations include:

- I/O
- Display and hardcopy
- Type queries
- Equality queries (default equivalence function)
- *Copying (multiple semantics)*
- Mutability and locking control
- Environment queries (version, source code, object code, documentation)
- Inspecting and editing
- Memory management (to be specified)
- Persistence (to be specified).

Table 3-2 presents the method signatures (i.e., names, input types and output types) for *Object*. Table 3-3 expands the signatures to include documentation of each method.

3.3.2 Class

A *Class* describes an object class. Each class has a corresponding unique *Class* instance. Table 3-4 shows the method signatures and Table 3-5 shows the definitions.

Table 3-2: *Object* Method Signatures.

Name	Input Types	Output Types
newLike	()	Object
copy	()	Obj:Object
deepCopy	()	Obj:Object
shallowCopy	()	Object
deepenShallowCopy	(Object)	Object
classOf	()	Class
isSame	(Object)	Boolean
isKindOf	(Class)	Boolean
species	()	Boolean
isSpecies	(Object)	Boolean
isUnique	()	Boolean
isImmutable	()	Boolean
changeToImmutable	()	Status
isLocked	()	Status
lock	()	Status
unlock	()	Status
destroy	()	Status
deepDestroy	()	Status
reset	()	Status
isEqual	(Object)	Boolean
display	(Displayable)	Status
inspect	()	Status
edit	()	Status
describe	(OStream)	Status

Table 3-3: *Object* Methods Definitions

Name	Input Types	Output Types
newLike	()	Object
Make a new instance of the same class as self and load with default values.		
copy	()	Obj:Object
The "natural" copy. Make a new instance of the same class as self. Copy "shallowly" those slots that expect address equality or whose objects are immutable. Copy others "deeply." Further, *copy* is free to change the internal structure as long as self->isEqual(Obj) would be True if executed and value of a mutable object is not stored by copying the mutable object's address.		
deepCopy	()	Obj:Object
Make a new instance of the same class as self, copy non-pointer values from self to Obj, and set all internal pointers of Obj to deepCopy's of what self points to.		
shallowCopy	()	Object
Make a new instance of the same class as self, copy non-pointer values from self to Obj, and set all internal pointers of Obj to point where the internal pointers of self point.		
deepenShallowCopy	(Object)	Object
Turn a shallowCopy'ed instance into a deepCopy'ed one. Method used by deepCopy.		
classOf	()	Class
Return the Class instance corresponding to the class of self.		
isSame	(Obj:Object)	Boolean
Return True iff self and A are Obj are identically the same object (same address).		
isKindOf	(c:Class)	Boolean
Return True iff self is an instance of the class *c* or its subclasses.		
species	()	Boolean
Return the class (often abstract) in the hierarchy that defines the semantics of the state of self.		
isSpecies	(Obj:Object)	Boolean
Return True iff self and Obj are of the same species.		
isUnique	()	Boolean
Returns True iff self is the only object that can obtain its current state.		
isImmutable	()	Boolean
Returns True iff the apparent state of self can never (again) be changed with the object interfaces. Once self->isImmutable() returns True, it should always return True.		
changeToImmutable	()	Status
If possible, change the state of self so that self->isImmutable() will return True.		

Table 3-3: *Object* Methods Definitions (cont'd)

Name	Input Types	Output Types
isLocked	()	Status
Return True iff the apparent state of self cannot currently be changed with the object interfaces.		
lock	()	Status
If possible, change the state of self so that self->isLocked will return True.		
unlock	()	Status
If possible, change the state of self so that self->isLocked will return False.		
destroy	()	Status
Destroy self.		
deepDestroy	()	Status
Recursively destroy self and all its contained objects where what is "contained" is class-specific.		
reset	()	Status
Reset state of self to the class-specific default.		
isEqual	(Obj:Object)	Boolean
isEqual forms the species-specific "natural" equivalence relation between instances. It returns True iff self and Obj are "equal".		
display	(D:<Displayable>)	Status
Display self on *D*.		
inspect	()	Status
Initiate the interactive inspection of self.		
edit	()	Status
Initiate the interactive editing of self.		
describe	(Out:<OStream>)	Status
Describe self on Out.		
name	()	String
Return the name of the class referred to by self		
directSuperclasses	()	Set
Return the set of class instances of the direct superclasses of class referred to by self.		
superclasses	()	Set
Return the set of class instances of the superclasses of class referred to by self.		
directSubclasses	()	Set
Return the set of class instances of the direct subclasses of class referred to by self.		
subclasses	()	Set
Return the set of class instances of the subclasses of class referred to by self.		

Table 3-3: *Object* Methods Definitions (cont'd)

Name	Input Types	Output Types
members	()	Set
Return a set of slot and function descriptors describing the class referred to by self. A descriptor should include name, slot or function, type, signature (if describing a function), access restrictions, file name where function is defined, whether static or class-allocated, etc. Building descriptors requires parsing code (or perhaps using symbol table information).		
headerFile	()	FileName
Return the file defining the class referred to by self.		
version	()	Version
Returns a version number or name for the class referred to by self.		

Table 3-4: Method Signatures

Name	Input Types	Output Types
name	()	String
directSuperclasses	()	Set
superclasses	()	Set
directSubclasses	()	Set
subclasses	()	Set
members	()	Set
headerFile	()	FileName
version	()	Version

Table 3-5: Method Definitions

Name	Input Types	Output Types
name	()	String
Return the name of the class referred to by self		
directSuperclasses	()	Set
Return the set of class instances of the direct superclasses of class referred to by self.		
superclasses	()	Set
Return the set of class instances of the superclasses of class referred to by self.		
directSubclasses	()	Set
Return the set of class instances of the direct subclasses of class referred to by self.		
subclasses	()	Set
Return the set of class instances of the *subclasses of class* referred to by self.		
members	()	Set
Return a set of slot and function descriptors describing the class referred to by self. A descriptor should include name, slot or function, type, signature (if describing a function), access restrictions, file name where function is defined, whether static or class-allocated, etc. Building descriptors requires parsing code (or perhaps using symbol table information).		
headerFile	()	FileName
Return the file defining the class referred to by self.		
version	()	Version
Returns a version number or name for the class referred to by self.		

3.3.3 Collection

A *Collection* represents an object that "collects" or "contains" other objects. The *collection* class specializes to many classes including Bag, Set, Relation, Function, and Image. A raw *Collection* has axioms that allow for remembering duplicate elements and insert order. A *Bag* adds an axiom that makes it impossible to remember insert order. A *Set* adds an axiom that makes adding an element many time the same as adding it only once. Other subclasses add axioms to restrict the types of elements that can be added (e.g., *Tuples* with instance-specific schema for relations) and vary the notion of equivalence between elements (e.g., a *IdentSet* would check the address of instances while an IntSet would check for the numerical value of instances or primitive datatypes.) Table 3-6 presents the signatures of a minimal set of methods for *Collection.*

Elements must satisfy `insertCompatible` to be valid arguments to `insert`. Other collections must satisfy `unionCompatible` to be valid arguments to `union`, `intersect`, etc. Many subclasses of *Collection* do not have to to check for argument compatibility since any argument that passes static type checking (or dynamic dispatching) will be acceptable.

A fundamental property of each collection is its equivalence function (*equivalentArgs*) for determining if two objects are equal for all purposes of the collection. A collection is best thought of as containing abstract elements rather than objects themselves (only when the equivalence function is object equivalence not a more general function of internal state). An element should be thought of as naming the abstract object equivalent to the instance. For example, a collection does not necessarily "return" the same instances (as chunks of memory) that are inserted, it can return any equal (under the equivalence function) objects. Also, any two sequences of operations differing only by substitution of equivalent instances and ending in a "isMember" or "isSubset" type of test will yield identical test results.

Equivalence relations allow collections to be extensible and highly efficient. For instance, suppose that "object value" is the equivalence relation for a certain set. Then, the collection does not guarantee that an object, once inserted, can ever be recovered. The class simply guarantees that an object of the same value can be recovered. Specialized collection classes may be then able to use efficient memory schemes (like arrays) to minimize storage and to speed computation. It can also considerably reduce the work required of a persistent storage system. Only values needed to copy objects (not objects themselves) have to be stored, thus eliminating the overhead of maintaining persistent UID's. The option for storing memory-objects is always available by making the equivalence function be #'EQ (or &a == &b). Cartesian product equivalence functions are used for relations and functions. This unifies numerous data types often treated more independently. Examples include

Table 3-6: *Collection* Method Signatures.

Name	Input Types	Output Types
equivalentArgs	(Object, Object)	Boolean
unionCompatible	(Coll:Collection)	Boolean
unionCompatible	(Primitive:<T>)	Boolean
insertCompatible	(Obj:Object)	Boolean
insertCompatible	(Primitive:<T>)	Boolean
isSingleton	()	Boolean
isEmpty	()	Boolean
isFinite	()	Boolean
isCountable	()	Boolean
isCountablyInfinite	()	Boolean
isUnCountable	()	Boolean
cardinality	()	int
cardinality1	()	Cardinal
hasMember	(Obj:Object)	Boolean
hasMember	(Primitive:<T>)	Boolean
disjoint	(Coll:Collection)	Boolean
isSubset	(Coll:*Collection*)	Boolean
insert	(Obj:Object)	Collection
insert	(Primitive:<T>)	Collection
remove	(Obj:Object)	Collection
remove	(Primitive:<T>)	Collection
remove1	(Object)	Collection
remove1	(<T>)	Collection
union	(In:Collection)	Out:Collection
difference	(In:Collection)	Out:Collection
intersection	(Collection)	Collection
relativeComplement	(Collection)	Collection
symmetricDifference	(Collection)	Collection
choose1	(Collection)	(Object Boolean Collection)
choose1	(Collection)	(<T> Boolean Collection)
map	(Collection, func,Class)	Collection
map	(Collection, func,Class)	Collection
mapc	(Collection, func)	void
mapc	(Collection, func)	void

numerical arrays, object arrays, hashtables, relational tables, and procedurally defined functions.

If the equivalence function is object identity (which we will call the object eq), then the actual instance inserted into the collection has to be retrievable/testable. If the equivalence function allows different objects with the same "values" to be equal, then there is no guarantee that the instance inserted is actually stored (e.g., a relational table will generally disregard an inserted tuple object after the tuple's field values have been extracted and stored in a relation-specific internal form). An equal-collection cannot rely on storing a mutable object as the sole memory of an insert as the object could mutate and no longer denote the same value it did at time of insert.

Persistent collections introduce further considerations (e.g., a persistent eq-collection can contain only persistent objects). A set of consistency rules and implementation possibilities are as follows:

- persistent eq-collection ⇒ elements must persist
- eq-collection ⇒ can use any object reference (e.g., persistent Object ID (OID) or pointer) for representing elements in working storage
- equal-collection and immutable element ⇒ can use any object reference or any equivalent data for representing the element in working storage
- equal-collection and mutable element ⇒ element must be "copied" into immutable equivalent data for working and/or persistent storage
- persistent eq-collection ⇒ must use OID for representing elements in persistent storage
- persistent equal-collection and immutable element ⇒ can use OID (if available) or data for representing element in persistent storage.
- persistent equal-collection and mutable element element must be "copied" into immutable equivalent data for working and/or persistent storage

3.3.4 Bag

A *Bag* is a collection of order independent elements with meaningful duplicates. The method signatures are unchanged from those of *Collection*.

3.3.5 Set

A *Set* is a collection of order independent elements with duplicates meaning the same as only one. The method signatures are unchanged from those of *Collection*.

3.3.6 PointSet

A *PointSet* is set of points in an n-dimensional Euclidean space.

3.3.7 PointSpace

A *PointSpace* is an n-dimensional Euclidean space. Abstract points in PointSets can be arbitrarily subtracted to form vectors. Vectors can be added to points. Elements of base PointSets have no scalar multiplication operator and only barycentric addition (coefficients sum to one). Elements of base PointSets are abstract, without methods for coordinate representation, default point frame, or default vector basis. (They can be, for instance, abstract handles to locations in the real world.) The base PointSet class is specialized in different ways to get many important classes. Through different types of specialization, instances of subclasses may have:

- Specific embedding dimensions (dimension of the PointSpace containing the PointSet).
- Specific embedded dimensions (manifold dimensions of the PointSet itself).
- Points with associated coordinate representations (with respect to a distinguished frame per instance).
- Vectors for elements. There is a distinguished origin per instance, and new operators are defined. These include multiplication by a scalar, vector addition, inner product, and (for three-dimensional vectors) cross-product.
- An interpretation as a specific physical concepts (e.g., physical space, time, space-time, image pixel locations, window pixel location, viewport space, screen locations, color space, spline function space).

Such specializations include: Frame, Basis, Interval, Region, Volume, TimeInterval, DurationInterval, ScreenFrame, ColorBasis, TimeFrame, VectorSubspace, CoordinatePointSpace CoordinateVectorSpace, and TriangularFacet.

A potentially useful way to specify the class of points that are elements of a PointSet is to identify singleton sets with points. With this, operations automatically generalize from points to sets of points and there is no need to define classes both for sets of points and for points themselves. This meriology is the most common method of treating schemas in the database literature. Also, Wand has suggested it as part of a general approach to modeling real things.

A request for a change of basis (change of frame) includes the new basis (frame) for the coordinate representation. This is different from a transformation operation that keeps the same basis or frame (if one is used at all). Dynamic type checking should

Table 3-7: Additional *Relation* Method Signatures

Name	Input Types	Output Types
join	(Relation)	Relation
relationalDivide	(Relation)	Relation
select	(Expression)	Relation
select	(Function)	Relation
projectOn	(Set)	Relation
projectOff	(Set)	Relation
thetaJoin	(Relation)	Relation
semiThetaJoin	(Relation, Expression)	Relation
attributes	(Relation)	Set
addAttribute	(Relation, Attribute, Object)	Relation
removeAttribute	(Relation, Attribute)	Relation

not allow operations on mismatched frames (or, if possible, will automatically put two coordinate representation into the same frame via appropriate change of frame calculations).

3.3.8 Relation

A *Relation* is a set of tuples, all having the same domain (schema). A tuple is a mapping from an index set into values. The index set is a set of attribute objects, not simply a set of consecutive numbers starting at zero or one. Table 3-7 gives the additional method signatures for a *Relation.*

3.3.9 Function

A *Function* is a relation in which the values of the **dom** attribute set uniquely determine values of the range attribute set. Every point in a **dom** set maps to a point in the **ran** set . Further, every point in the **ran** set corresponds to one or more points in the **dom** set. **dom** and **ran** belong to (possibly larger sets) domain and range. Table 3-8 gives the additional method signatures for a *Function.*

3.3.10 Tuple

A *Tuple* is a function with a domain that is a set of attributes.

Table 3-8: Additional *Function* Method Signatures

Name	Input Types	Output Types
domain	()	Set
range	()	Set
dom	()	Set
ran	()	Set
isPartial	()	Boolean
isOneToOne	()	Boolean
isOnto	()	Boolean
isInvertible	()	Boolean
evaluate	(Object)	Object
evaluate	(<T>, <T>, ...)	Object
atPut	(Object, Object)	Boolean
compose	(Function)	Function
composeClass	(Function)	Function
composeCompatible	(Function)	Function
invert	()	Relation
restrict	(Set)	Function
overridingUnion	(Function)	Function
curry	(Function)	Function
dispatch	(Function, Function, ...)	Function

3.3.11 RealFunction

A *RealFunction* is a function with range a subset of the real numbers. It maps all unary and binary numerical operators (e.g., log and $\times$) onto the binary RealFunction operators.

3.3.12 Array

An *Array* is a function with dom a subset of $[i : j] \times [k : l]$ for integers i, j, k, l.

3.3.13 Image

A *Image* is an array with extra properties related to its collection or creation.

3.3.14 DiHypergraph

Name	Input Types	Output Types
isPolytree	()	Boolean
isTree	()	Boolean
nodes	()	Set
arcs	()	Set
parentNodes	(Object)	Set
parentNodes	(Hyperarc)	Set
childrenNodes	(Object)	Set
childrenNodes	(Hyperarc)	Set
parentArcs	(Object)	Set
parentArcs	(Hyperarc)	Set
childrenArcs	(Object)	Set
childrenArcs	(Hyperarc)	Set
asRelation	()	Relation

3.3.15 BayesNet

Name	Input Types	Output Types
prior	(Object)	BayesArc
bel	(BayesArc)	BayesArc
asBayesArc	()	BayesArc

3.3.16 Hyperarc

Name	Input Types	Output Types
parentNodes	()	Set
childrenNodes	()	Set
asFunction	()	Function

3.3.17 BayesArc

Name	Input Types	Output Types
multiply	(BayesArc)	BayesArc
instantiate	(Tuple)	BayesArc
instantiate	(Tuple)	BayesArc
condition	(Set)	BayesArc
divide	(BayesArc)	BayesArc
marginal	(Set)	BayesArc
multAndMarginal	(BayesArc, Set)	BayesArc

3.3.18 Attribute

Attributes are elements of tuple domains. They are objects in many presentations of relation databases. They are random variables in condition probability densities and Bayesian networks. *Attributes* can have associated domain constraints, refer to measurable quantities, have units, and have physical interpretations (like row of a display screen).

3.3.19 Unit

Subspaces, frames and bases can be specialized to allow individual elements to be measurable with respect to certain known units. Distinguished (const and often static) instances of the *Unit* class include: meter, kilogram, second, coulomb, radian, degree (absolute), and monomial combinations (e.g., inch, m^2, and w/sr-Hz).

3.3.20 Dimension

Each *Unit* instance is associated with a type of measurement quantity (or dimension). Instances of the *Dimension* class include: length, mass, time, charge, plane-angle, and temperature (with many dozen more).

Section 4

User Interface

4.1 Introduction

A major objective of the Image Understanding Environment User Interface (IUEUI) is to give users flexible, simple, and powerful tools for exploring data, algorithms, and systems. Another fundamental objective is to create an interface which will be supported by ongoing and future developments in the software world at large. To achieve this, we want to capture the critical functionality of our domain in a small number of objects which are built on top of existing interface packages and interface construction toolkits. Efficiency and long term extensibility will increase by implementing features on the correct level. The same objects which will be used to implement the user interface can be specialized for users of different skill levels and objectives. This is all critical for the long term use of our environment because we can depend on dramatic changes in interface devices, voice input, video publishing, network interfaces, hypermedia databases, tools for cooperative work and communication. These are all things we want the environment to take advantage of as they are developed.

The Interface of the IUE is described in terms of three levels (Figure 4-1). The **Graphics Level** is the underlying "machine independent" package for basic display and graphic operations and telling the screen what to do. Examples would be X and Postscript. Machine independence is somewhat relative, so for now, this level can include other packages as long as they support similar functionality. The **Interface Kit Level** consists of existing packages for the creation and rapid prototyping of user interfaces and related tools on top of graphics level software. Examples are such things as Interviews, TAE, NeXTSTEP. This also must include use of tools found in the selected software development environment such as editors and debuggers. The **Image Understanding Environment User Interface (IUEUI) Level** consists of the objects in the user interface. This includes such things as object displays, plotting displays, several types of browsers, and structures for describing the interface context. The IUEUI consists of a small set of objects which can be freely combined for very powerful results. The specifications of these objects is relatively independent of the

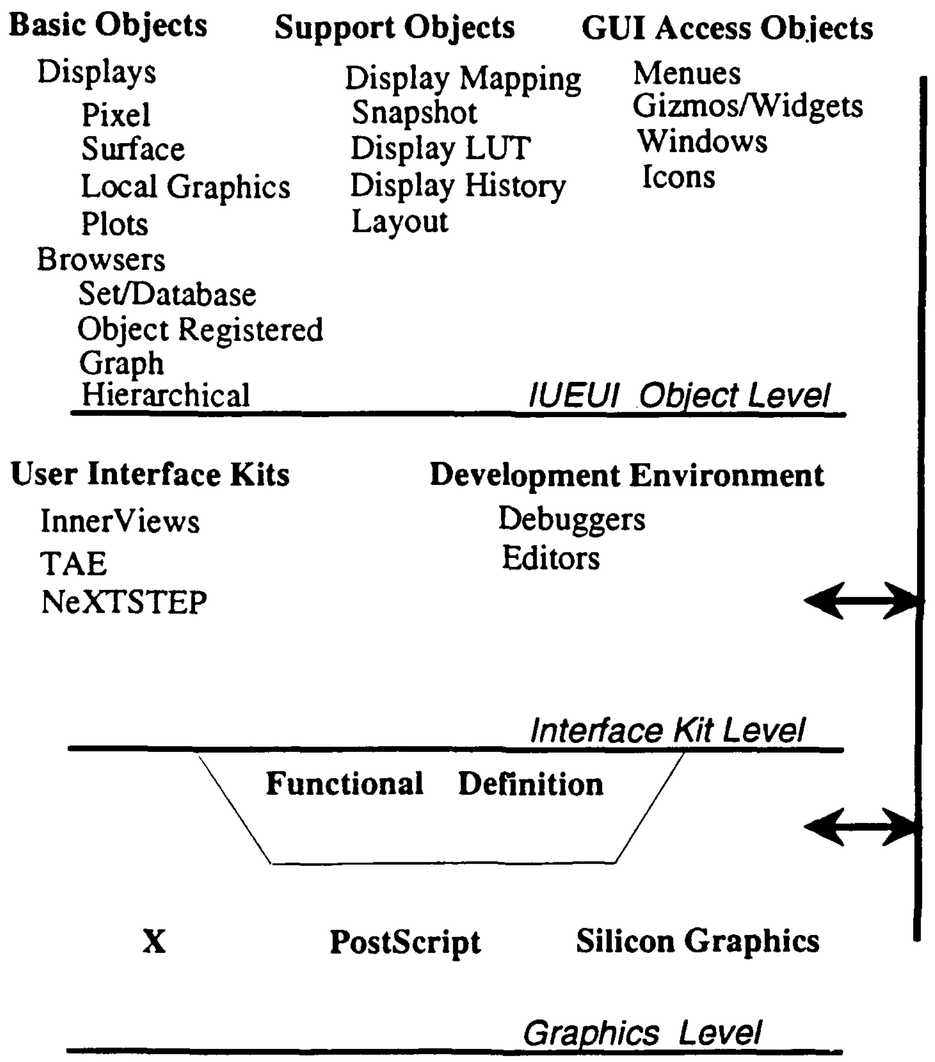


Figure 4-1: Levels of Interface Implementation.

other two levels although it will be required to show in the programmers manual how the functionality of the IUEUI objects are realized at the other two levels.

4.1.1 Paradigms

Some of the different interface paradigms we have discussed are:

- Subject/View
- Data/Flow
- Display methods as integral parts of objects

These are not disjoint in our design and each reflects critical capabilities for the IUEUI. An object-oriented methodology is essential for the interface because we don't want users to have to use literally thousands of different functions depending on the type of object they are trying to display, browse, or inspect. This is often a problem with C-based environments because users need to specify the type of value and the type of object. At the same time, we need to provide explicit, powerful tools so users can control all aspects of data visualization at anytime. We often want to display the same object in several different ways (displaying the gradient as a vector field, or mapping the different components of the gradient onto different color bands). It is not sufficient to let the user simply tell an object to display itself and then let the object "decide". It is essential that users be able to view objects in a completely controlled manner and that there are intelligent defaults reflecting common usage for less experienced users. Dataflow capabilities are necessary to allow a user to create a complex process by interactively specifying the combination of other processes and to monitor their execution.

4.2 Required Functionality

This section describes the necessary functionality for the user interface based upon the union of attributes found in several different existing IUEs and, in some cases, their deficiencies.

- **Object Display Control** We want the user to have complete control of how spatial objects (images, surfaces, features, networks, pyramids, etc.) are viewed with maximal flexibility and ease of specification. The mapping between a spatial object and a display window needs to be explicitly represented by a Display Mapping Object that can be manipulated, set, and saved. There are two basic aspects of this. One concerns how positions on the object get mapped onto positions in a display windows. For this, users want to control such things as zooming, panning, perspective, warping, and other geometric operations. For now we refer to this as the Position Mapping. The other aspect concerns how values of an object get mapped onto display window values such as intensity and color (sometimes text, icons, and graphics). For now we refer to this as the Value Mapping.

The distinction between the position and the value of a spatial object can be complicated and a user needs to be able to access any attribute of an object and to display it as he wishes. For example, a discrete curve can be viewed as a mapping from integer indices onto 2D positions with respect to an image coordinate system. When I overlay the curve on top of an image, I am mapping these 2D positions along the curve onto window positions using the same position mapping that was used for the display of the image. I also would like to control the color/intensity of the display at these points based upon registered values associated with the curve (such as curvature). For example, a user might want to display an intensity image in 8-bits of green intensity and then overlay extracted curves on top of this with the display of curvature values along the curve mapped onto 8-bits of red intensity. And to say this at least as directly as the previous sentence.

- **Virtual Objects** When display transformations are applied to an object, it should not involve creating a new object: only the *display in a window is* generated. An example is manipulating the underlying color look up table to perform a thresholding operation. In this case, there is no thresholded image object produced, only that which is displayed in a window. This goes by many names in different systems such as Pixel Mapping Functions, Dynamic Color, Generalized Color Look-Up Tables. It includes operations such as thresholding, histogram equalization, fitting to a linear display range, overlays, and others. The display buffer is a short term memory for a view of a displayed object: we should perhaps provide routines to use this directly. Some work on the Lisp Machines would implement image processing operations by bit-blitting.

- **Commands and Intelligent Defaults to Deal with 8-bit, 24-bit, 32-bit displays**: The interface should be able to deal with different types of display devices, taking into account how deep the display buffer is.

- **Display Overlays**: It is important to be able to display extracted features and values overlaid on top of images (and other objects), as in displaying a vector field on top of an image, or in displaying extracted edges and junctions. The overlays can occur with respect to the display window (annotating an image with text) or with respect to the displayed objects (marking a displayed surface with features that occur upon it). This is one aspect of the specification of multi-object displays.

- **Linking Displays and Browsers:** Viewing transformations can be concatenated through links between displays. The view (the mapping between a spatial object and a display window) in one window can be concatenated with the view specification in another. A common example is using one window to zoom onto the display in another or using one window to display a selected portion of another. Panning and Zooming are so common they will probably be directly supported as a default or through a system level menu. It is also useful to have links between browsers, such that when an object is selected in one browser, the attributes of the selected object can be viewed in another.

- **Interface Context:** Information describing the current context of the interface state is used for intelligent defaulting. This would include things such as the current window, the current mapping, established links between windows, the thickness of lines in graphic overlays, the current displayed objects, the layout of windows and browsers on a screen, and several other things. This would be an extension to the underlying context provided by the graphics level. These contexts can be saved and read. Selection of information through browsers also has a context for intelligent defaulting. This includes such things as the most recently selected object, links between browser windows. This is especially important when interactive browsing is used for sequences of queries over a database.

- ***Interactive Command Language:*** All display actions involving objects can be specified through an interactive command language. This should have intelligent defaults and abbreviations (such as displaying to the current window if none is specified). These commands should also be usable in code for creating scripts and general display routines. It is not necessary that all interactions take place through this command language (Some will be invoked by menus and special keys and refer to the current display context). The Interactive Command Language is especially important because it provides a functional specification of the entire interface.

- **Process Monitoring:** Monitoring the execution of a task; visualizing processing at the current locus of processing. This should be performed using the general browser class over a task database.

- **Animation Tools/Operations:** dumping window or screen output to video tape; cycling through a sequence of displays or displays written out to file; cycling through a sequence of displays; use of double buffering in the display buffer.

- **HardCopy Tools/Operations:** dumping window or screen output to paper, slides, overhead transparencies. [**Note:** should we make PostScript compatibility a requirement?]

- **Interactive Command Buffer**: The user can type-in display commands in an interactive buffer; he can cycle through commands, he can perform window-based editing operations on commands in the command buffer and then specify their re-execution. A good model is the ease of use with the Lisp Listener. We need the same functionality for interface and object interaction operations even in a C-based environment.

- **Synergy of Interface and Development Environment**: When moving from the debugger and editors for code development to the display and browsing operations of the interface, it should not feel like starting up completely different processes.

- **Object Interaction** Displays are also used for interacting with spatial objects in order to access values in them, move them around, and apply operations to them. In interactive processing, when the user clicks on the display window, the position in the window and the current object and the current object value are saved. The current object can be explicitly specified or taken from context. Disambiguation may be required if there are multiple overlapping objects. The user may be required to use a label plane (an image of pointers to objects which occupy a given position) or use geometrical data base operations in the IUE. Both are potentially expensive and don't reflect operations specific to the IUEUI but are general IUE spatial data base operations that can be accessed through the IUEUI. It is sufficient that the interface is able to return the selected object(s) and object position from the object display mapping.

- **Graphics and Text Overlays**: This involves writing and graphic drawing (both 2D and 3D) with respect to a display. This is especially useful for slide creation, documentation, and data generation. It has several modes that need to be distinguished. Sometimes we want to do this with respect to the window in which a spatial object is displayed; sometimes with respect to the displayed spatial object or object coordinate system (perspective view of an object model); sometimes we want the generated graphic display to generate and instantiate corresponding IUE objects; and sometimes we want to refer to the entire screen on which several display may be present, as in connecting features in different windows and browsers by arrows or annotations.

- **Default layouts for windows and browsers**: The desired layout of windows and browsers can be saved and be available to a user when he starts using the IUE. Several different such arrangement can be stored and brought back for different situations under user control.

- **Simplified Access to Interface Objects**: The IUE should provide simplified, interactive access to the interface objects found in GUI Kits. Such things as sliders, knobs, buttons, text input/output fields, menu creation and personalization, and icons.

- **Display History**: The sequence of display or browsing actions for a particular window or browser are saved and can be reaccessed and used for creating animations.

- **Types of Displays to be supported**: Image, Edge/Contour, Vector, 1D Plot, 2D Plots, ND Plots, Features, Surface (grid and rendered), Volumes, Color Images, Sets/Databases of feature objects [be able to refer to selected attributes], Image Sequences, Animations of Image Sets, Gray scale Images, Metrically embedded graphs (graphs showing relations between extracted image features).

- **Interactive Object Creation (Draw Objects)**: It should be possible to create object interactively. This is useful for creating sample idealized data for testing and development. Interactive Object Editing and Creation should also be supported.

- **Updating Displayed Objects**: In some cases we want a display or a browser to be updated when an object changes (such as a browser for a database of executing tasks).

- **In-code Functions for prompting user for unspecified information and other interface actions**: enable user code to refer to interface actions. Examples: A routine can specify that a particular value is to be queried from a user interactively or set from some gizmo/widget, or that a particular browser field is to be updated when some state occurs or a process completes execution.

- **Multiple Object Displays**: A spatial object is viewed through a display window by a mapping from the spatial object onto the display window. It is important to be able to map several spatial objects onto the same window at the same time or through the incremental creation of a display. Examples are for such things as mapping different images in a stereo pair onto different color planes; overlaying extracted features onto an image (or arbitrary surface). For operations such as overlays, transparency, Color, Cycle, flashing, display buffer animation.

- **Mensuration tools**: rulers, grid overlays, flashing them; orientations of rulers under zoom-links between windows; and cursor type and the use of multiple cursors. Whether the measurements are being made only relative to the 2D display window or with respect to the spatial object, these can be built on top of the basic interface capabilities and the display of IUE objects (in particular, the interactively functionality of the display object and IUE objects such as bit-mapped regions, line-objects).

- **Incorporating Hardware Accelerators**: The interface (and the IUE in general) should be able to deal with different hardware accelerators and a distributed computing network.

- **Interactive programming tools**: Several questions remain to be answered: Can these be as in Khoros (an image processing system with data flow programming environment freely distributed by University of New Mexico)? Can these be developed from the graph browser and icons? How are DataFlow transformations handled?

- **Device Menu Shortcuts (Bucky Windows)**: Many menu operations should be mapped to combinations of key and mouse button processes. The user should be able to modify and extend the default mapping.

- **Access to and Integrated use of Established Visualization Packages**: There now exist several data visualization products (Plotting package in Mathematica, packages from Precision Visuals). We should select those which are relevant and supply interfaces to them. In fact, we will find ourselves recreating this functionality (especially for plotting). There are, however, problems with data type compatibility, speed, selection tools to get exactly what we want displayed from these packages in the IUE. Much of the interactivity may not be obtainable and the feel may be considerably different than the IUEUI.

4.3 IUE Interface Objects

We break the IUE interface objects into three basic classes (See Figure 4-2). The first class consists of displays and browsers. These are the basic tools for viewing an object and inspecting its symbolic attributes and relations (There are many commonalities between these objects that suggest a meaningful and general IUE interface object). The major portion of what a user does with the interface will be based upon these objects. The second class are the objects commonly used in the supporting user interface mechanisms provided by the graphics and toolkit level (menus, widgets, icons, etc.) but with simplified commands so they can be manipulated directly by IUE users. The third class are support objects for such things as describing the current interface context, the mappings from an spatial object onto a display window, links between IUE interface objects, animation files, and several other things. Many of these are not necessarily objects, but common data structures.

4.3.1 Object Display and Browsers

- **Object Displays**: (See Figure 4-3) This is for viewing objects which have coordinate systems associated with them and mapping them onto a 2D display.

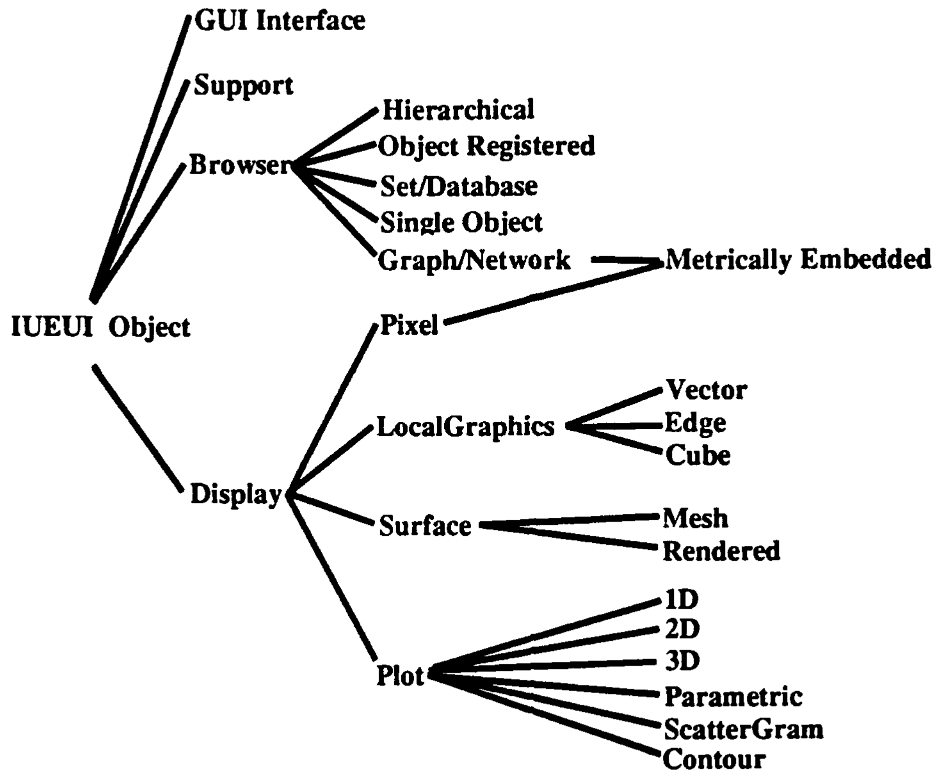


Figure 4-2: IUEUI Object Hierarchy.

It includes such things as images, curves, regions, object models, surfaces, vector fields, etc. They support several types of operations for controlling the mapping of an object to be viewed in a window and for interacting with a displayed object. There are several subclasses of displays that will appear to the user to occur in the same type of window. They are primarily distinguished by the types of methods they understand and all inherit a large number of similar methods from the general display class. For example, the overlay method means something different in the context of a surface display than in the context of an image display. The **pixel** display class is for viewing images and image registered features. The **local graphics** display class displays objects by mapping their values onto graphic objects such as lines and cubes. Examples are displaying vector fields and edges. The **surface** display class is for displaying objects that get mapped onto mesh or rendered surfaces. There are several different types of **plot display**: 1D, 2D, 3D graphs, histograms, scatter grams, perspective views of functions and tables. **Note:** We may implement plotting windows from existing packages (e.g., Mathematica). It will be complicated to extend or integrate such plotting/visualization packages with the other capabilities which will definitely be associated with display-windows (such as object interactivity and selection, function application, window-linking). We may find that we can live with this or we will need to implement some subset of these plotting packages directly in our interface.

- **Browsers**: These are used for actions such as queries over set of objects, determining and inspecting relationships between objects, process monitoring, and inspecting values in an object. There are 2 different types of browsers: **Field Browsers** and **Graph Browsers.**

  Field Browsers consist of a regular array of fields. Fields can be filled with text, icons, colors, colored text, text in particular fonts. Fields can have actions associated with them when they are selected or a user changes the values in them. We distinguish between four types of Field Browsers which inherit from the general Field Browser class:

  - **Set/Database Browser**: This is presented as an array of fields. Each row of fields corresponds to selected attributes of a particular object and each column corresponds to common attributes over the set (or database) of objects. An example would be browsing the database which describes the current active object in the IUE to find the most recently created image from some operations (See Figure 4-6).
  - **Single Object Browser**: Each row corresponds to the value of an attribute for an object. This is used for inspecting a single object (See Figure 4-7).
  - **Hierarchical Browser**: Useful for text based inspection of graph structures and trees. When an item is selected, the related items (along some relational dimension) are displayed in the next column (See Figure 4-8). [**Note**: A good example is the directory browser on the NeXT machine].
  - **Object-Registered Browser**: This contains values extracted from a spatial object, such as the intensity values in some square neighborhood of an image. Depending on the dimensionality of the object (or relationships between component objects), this can be presented as a 1D array, a 2D Array, or multiple 2D arrays and describes curves, images, image sequences, pyramids. There are restrictions on whether it is possible to interactively change values in the fields of an array browser. It should be possible to apply operations directly to the values in the array browser to see the effect of an operation in a restricted neighborhood of an object (See Figure 4-5).

- **Graph Browsers**: These are for the display of graphs and networks, generally representing an object as a node and links to describe relations to other objects. Nodes are similar to fields in field browsers and can be filled with text, icons, colors, colored text, text in particular fonts. Nodes can also have actions associated with them when they are selected or a user changes the values in them. Links can also be colored and selected. A typical use would be for the display of a constraint network (See Figure 4-9). **Note:** For complicated relationships or large sets of objects, these can become very complicated and we may need a way (e.g., GRASPER) for segmenting nodes and links into spaces].

An important type of graph and graph browser is a **metrically embedded graph** wherein the nodes (and perhaps links) are restricted to occur at positions with respect to a coordinate system. This type of graph inherits properties from both the Graph Browser and a general spatial object which can be viewed in a display window. An example would be an image registered network which describes potential links between extracted features for displaying grouping operations. An important attribute of metrically embedded graphs is that they can be viewed as an object display for operations such as zooming and having access to the underlying context in an image. [**Note**: We may also want to distinguish tree graph browsers.]

4.3.2 Simplified access to GUI objects

- **Gizmos and Widgets**: The IUE should provide simplified, interactive access to the interface objects found in GUI Kits- such things as sliders, knobs, buttons, text input/output fields, menu creation and personalization. This will involve commands for creating gizmos and widgets, for positioning and scaling them, for attaching them to parameters, for reading and writing to them. An example would be creating a slider and then getting values for an interactive thresholding operations from it.

- **Menus** The IUE should provide simplified interactive access to menus in the GUI kits. This involves being able to extend menus, create pop-up menus, associate actions with menu items. A critical design task is deciding what goes into system level menus and how they are organized

- **Icons** The IUE should provide simplified interactive access to icons in the GUI kits. These may not be flexible enough for our needs. [**Note**: Icons in Khoros representing processes can be connected together to form a graph for algorithm creation and process monitoring. We may be able to obtain the same functionality by using a graph browser with icons at nodes.]

4.3.3 Support Objects

There are also several objects that are used and manipulated as part of the interface that we will refer to:

- **Display-Look-Up-Table**: A generalization of a color look up table that describes how to map object values onto screen values. It can also include functions.

- **Object-Display-Mapping:** A structure which describes the mapping from an object onto a display. This includes both the position and values of how the object is displayed and a reference to a particular Display Look Up Table.
- **Object-Browsing-Mapping:** A structure which describes the mapping from an object or database onto a browser.
- **Object Display Links:** A structure which describes the concatenation of a display or browsing operation between IUE interface objects. Thus a link between display windows wl and w2 with an associated zoom and pan would display an object in wl with wl's object display mapping and then display the same object in w2 by concatenating onto the object display mapping for wl, the specified zoom and pan operation.
- **Interface layout:** A structure which describes the object instances in a particular instantiation of the interface. Users may prefer different interfaces (arrangement and instantiation of the basic IUEUI objects) depending on the task or level of sophistication.
- **Display Context:** A structure which describes current context for a display. Such things as the current window, the current object, the current object display mapping, the current display command, the current mouse-selected object position and value, and others. Display operations can use defaults based upon these.
- **Browse Context:** A similar structure for browsing operations. Such things as the current browser, the current data base, the query history, and others.
- **Display Snapshot:** What is produced when the current display is written to a file. It is just what appears on the screen and not the actual objects.
- **Animation File:** A sequence of display snapshots.

4.3.4 Interface to Arbitrary Interface Devices

In the development here, we are assuming a very limited set of interface object: a mouse and a keyboard. By the time the environment is commonly adopted, there will be a much wider array of objects such as: voice input, gestural recognition (data gloves), and perhaps virtual reality displays. What tools will we provide to interface such devices with the IUE? At this point, we are raising this as an important design issue. A rough cut was made at this earlier by suggesting objects

- Keyboard object
- Pointer object

- Mouse object
- Trackball object

with methods such as Open-Device, Get-History, Close-Device, Attach, Get-Status, Handle-Event. More extensive work will be needed in this area, perhaps requiring special IUE interface buffers and IUE access to Graphics Level event handling routines.

4.3.5 Total Synergy between IUE object methods and the IUE Interface Operations

Any of the operations that can be applied to objects as part of the IUE should also be applicable to object prior to displaying or browsing them. In the examples below, because the syntax for this is not specified, there will be some confusion. For example, how do I refer to the registered curvature values associated with a curve object if I want to display them? What is the query I use to find all curves greater than some length prior to displaying them? Using the interface requires use of objects and operations in the IUE, notably database operations and queries; referring to a particular set of attributes of an object; selecting a portion of an object; applying a transformation to object values; and operations for combining objects. Much of this comes for free in LISP. It may require building a special parser for non-interactive programming environments with an extensive library of functions. (**Note:** All IUE objects will also require browse and display methods.)

4.4 Object Display

The object display is for mapping an IUE object onto a 2 dimensional view with methods for controlling how this is done. It is a major object and is specialized into some different types of displays which have different inherited methods from the general display object. A display is also recursive in that it can consist of several displays. Performing and interacting with an object display takes place in an object display window. From the user point of view, an object display can be thought of as a window with different component displays in it.

4.4.1 Appearance

The layout for the object display window is shown in Figure 4-3 . It is a rectangular window with

- a title bar (which can be colored or patterned)

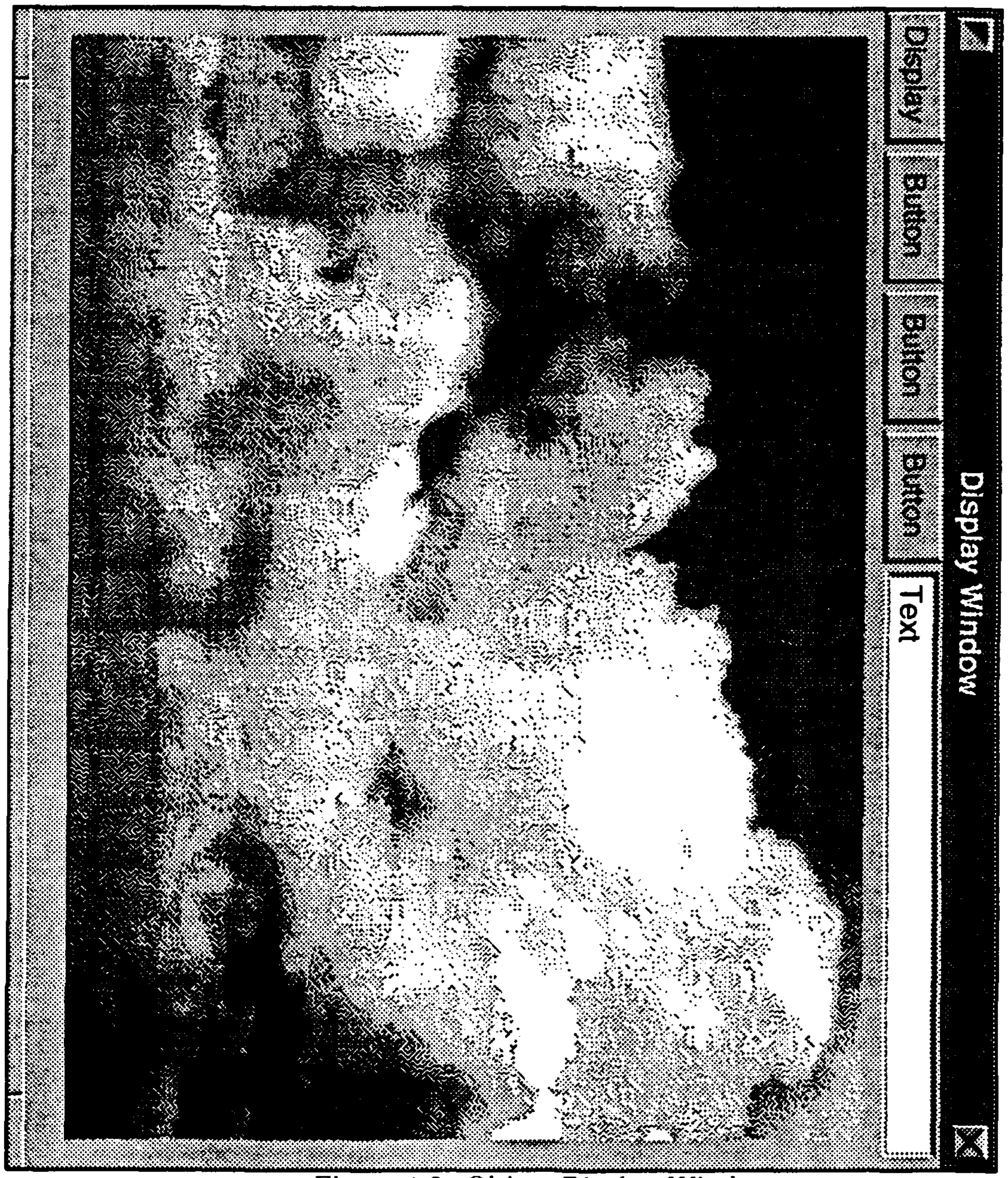


Figure 4-3: Object Display Window.

- a close box for closing the window
- an iconization-box for turning the window into an icon and saving it s state
- horizontal and vertical scroll bars for panning
- a resize box for changing the size of the window
- a status bar containing information about the current display object(s) and the last selected object location and value
- a button bar for actions such as display of the current browse selected object.

A window can be selected by clicking anywhere on it with a mouse. It then becomes bound to the variable named *current-window* for future displays and interactions. Thereafter, when the user clicks in the display portion of the window, he executes the interactive methods which defaults to displaying the current object location and value (the user can associate arbitrary interactive methods with mouse actions). The current window has a distinctive highlighting of the title bar as do any window to which is is linked. As an option, the user can hide other windows except for the current window and windows to which it is linked. The window can be repositioned by clicking and dragging on the title bar.

There are two other associated windows for interacting with the display in the current window:

- **Interactive Command Buffer**. This appears like a WYSIWYG text editor. Textual outputs can also be written to the Interactive Command Buffer. It has a vertical scroll bar for accessing previously written commands and allows operations like cutting, pasting, etc. It is something like a limited Lisp Listener for interacting with objects and displaying them. A nice feature it should incorporate is the use of shift-return in Mathematica: a command is only executed when the user hits shift-return. This makes it possible to create complex multi-line operations and also breaks the display command history into chunks of related actions that a user may want to access for later editing. (**Note**: It may be a Lisp Listener or something very similar if the development takes place in an interactive programming environment.)
- **Display Tool Box**. There are many familiar interactive controls for displays and visualization, such as interactively manipulating the object-value to screen-intensity function by interactively shaping the function, selecting color look-up tables, modifying color look-up tables, interactively building display commands using templates or command browsers, floating tool palette of interactive drawing tools, etc. The Display Tool Box is a menu of such tools, organized into functional groupings of interactive tools for manipulating the current display.

From a functional point of view, it allows potentially redundant access to the display methods without using the interactive command buffer. It is somewhat like the system control menu on the Macintosh and the system preference menu on the NeXT machine. In the Figure 4-4, the large buttons on the right are some of the different modes of interaction. The interaction field on the left has the corresponding interaction tools for a selected mode. The ones shown here are existing ones on the NeXT for manipulating the color look-up table. The order of the buttons in the tool box should be settable by the user. Users can also select particular interaction tools and have them occur as floating palette (in cases where the user want to interact with multiple tools from different sets of tools at the same time).

(**Note**: We probably need a similar menu for setting up system defaults and initializing characteristics of the IUE: initial layout, font selected, level of expertise, etc.)

Some of the tools sets that should be included in the tool box are:

- Interactive Selection and Modification of the the current color lookup table and display mapping function; cycling through different color look up tables
- Interactive Selection and modification of a display's attributes using browsers for the display window's attributes that can be used for changing attributes of the window and display; Browsers over the windows that the window has links with
- Interactive Display Command Creation such as Browsers for selecting existing position and value functions to be placed into commands; available options for the different display commands and their current defaults presented in template form
- User-created gizmos/widgets: sliders, buttons, knobs that the user has created
- Gizmo/widget arrangements of which a user is particularly fond
- Interactive gizmo/widget creation using gizmo selection-wells
- Sending displays to a printer
- Animation creation and playback

Note: We can either associate the command buffer and tool box with each display window or else have them automatically linked to the current window. The second alternative is simpler, though it may have some context problems since displays will always default to the current window. It may be best to default to one command buffer and one tool box, but to have their context shifted in terms of setting to different selected tools when different windows are selected.

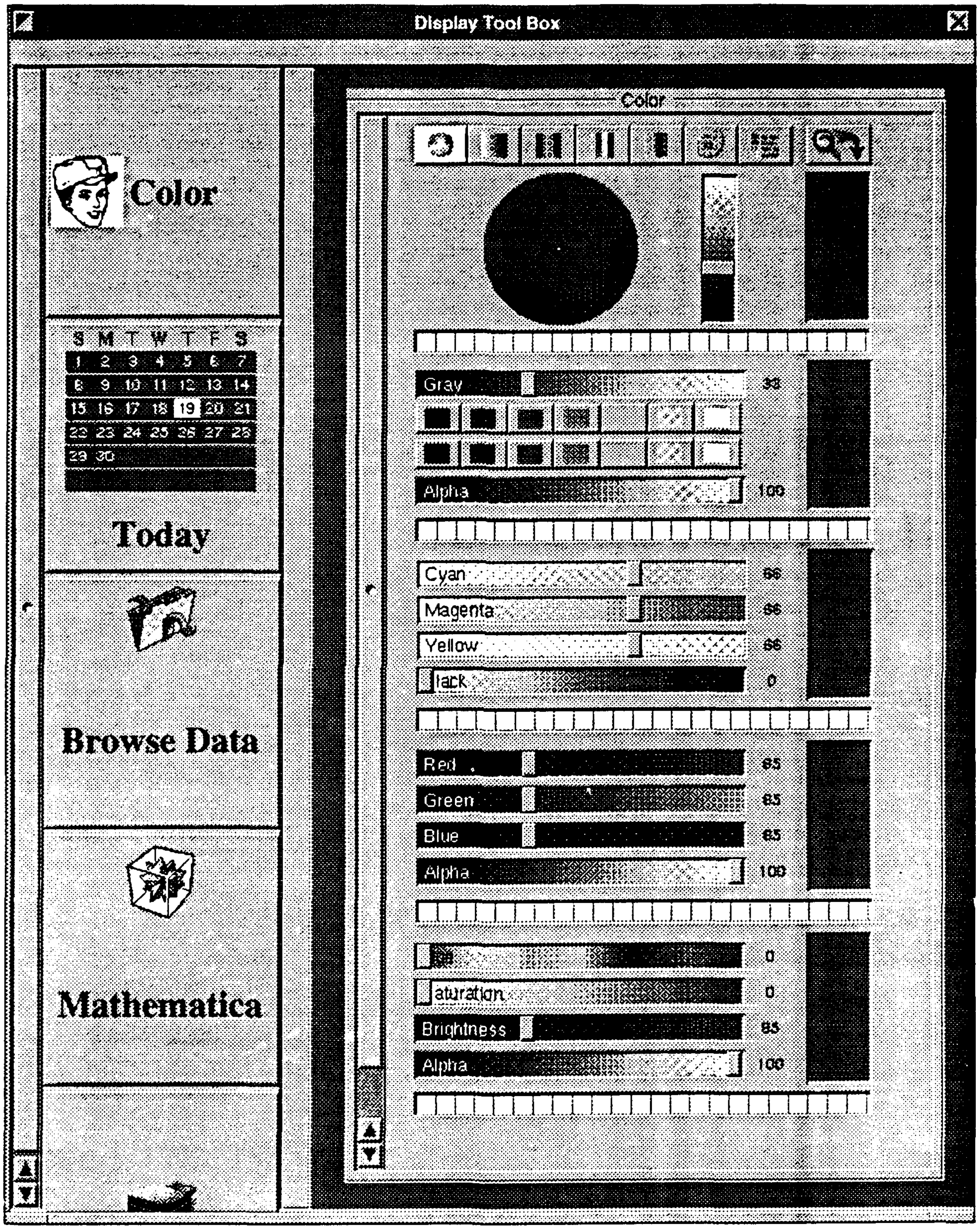


Figure 4-4: Display Tool Box

4.4.2 Attributes and Methods

Class-Name Object Display

Description An *Object Display*

Sources/existing implementations

Superior(s) IUE Interface Object

Component Class(s)

Associated Class(s)

Slots

Screen-Size: Size of window in screen coordinates. Returns a 2D integer vector (maybe a specialized struct)

Screen-Position: Position of upper, left-hand position of a display window in screen coordinates. A 2D integer vector (maybe a specialized struct)

Internal identifier: unique number associated with each display window

Name: Name of window: a string

Title-Bar-Color: Color of the title bar

Title-Bar-Font: Font Object

Cursor: Any of the set [cross-bar, cross-hairs, arrow, user-defined (**Note**: These should be definable as IUE Objects: lines with constraints between them mapped onto the overlay planes or bit-mask regions).

Children-Windows: A list of display-link objects describing displays which are linked to this display

Parent-Window: The display to which the display is linked

Current Object Display Mapping: An object view mapping object describing the current mapping of an object onto the display. This includes pixel scaling factors

Current Display Command: A set of strings for the current display command

Object-location-list: A list of locations selected by clicking the mouse on the display: it inverts the specified object view mapping (the default is the current one) to determine the location in the specified objects

Object-value-list: A list of values selected from an object using the object-location-list

Window-location-list: A list of locations selected by clicking the mouse on the display. The locations are in screen coordinates

Display History: A Database

Current Display Mapping Table: A display Mapping Table structure

Methods There are several methods for displays and they are organized into different groups (A good way to see the effects of these methods is to look at examples from the section on the command language):

Methods for Manipulating the Current Object display position mapping: This includes operations such as panning, zooming, perspective views, and warping. These are methods that control how positions in the specified object(s) get mapped onto a display window.

Methods for Manipulating the Current Object value mapping: These include operations such as overlays, mapping onto different color bands, transparency, and others. These are methods that control how values in the specified object(s) get mapped onto screen attributes such as color and intensity.

Methods for setting the current display mapping table: These include how to configure planes in the screen buffer for the display of color images; how many panes to use for overlays; particular functions and conditions to apply to object values prior to display.

Methods for Screen Attributes: These involve controlling attributes of the window the display is mapped onto and includes such things as position, size, attributes of the title bar, event handling for the mouse.

Methods for Links: Linking display transformations in different windows. Operations include creating links and associating position and value mappings with the links.

Methods for Interaction: These involve interaction and manipulation of displayed object(s) in the display. Operations include recovering object position and value from a mouse click, applying functions to selected objects, applying functions using selected information.

Methods for History: Methods to coordinate displays overtime, such as cycling through an image sequence, playing an animation of displays.

Methods for Graphics: These involve accessing display registered graphics packages for drawing lines, text, and other things. These occurs in four different modes: 1) relative to the window of the display; 2) relative to the entire screen; 3) the specified object or coordinate system; or 4) for instantiating IUE objects corresponding to the graphic displays.

Methods for printing and writing to file, animation: It should be possible to send the view of any object to a hardcopy print device or to files for later redisplay or printing.

Manipulating the Current Object Display Position Mapping

zoom (window-zoom)

Arglist x-zoom-factor y-zoom-factor

Return Type None

Description Specifies the scale of mapping from an object onto a window

Exceptions

pan (window-pan)

Arglist x-translation-factor y-translation-factor

Return Type None

Description Specifies the moving the position of a displayed object in a window

Exceptions

Matrix

Arglist Homogeneous Viewing Matrix

Return Type

Description

Exceptions

Position-function

Arglist Code Chunk

Return Type None

Description A chunk of code which computes a position from positions or values taken from the specified display objects. These are referred to as dummy variable by placing ".value" or ".position" after the object name (**Note**: what should be done if the values and positions are not just scalars?). As the display processing iterates over the specified object, the code chunk is applied to the specified values and positions prior to the display to generate a new position. In Lisp, these can be expressed as functional closures and are pretty easy to manipulate. In C, we may have to provide a compiled library of code chunks that can be used in this way or with a parser.

Exceptions

Warp

Arglist

Return Type

Description

Exceptions

Manipulating the Current Object Display Value Mapping

linear-mapping (linear-map)

Arglist object-value-min object-value-max screen-value-min screen-value-max

Return Type none

Description Will map the specified range of object values linearly onto screen intensity values

Exceptions screen-value-min and screen-value-max will have default values

values

Arglist a method for extracting values from a spatial object

Return Type none

Description

Exceptions

value-function

Arglist A chunk of code which computes a value from values taken from the specified display objects. These are referred to as dummy variable by placing ".value" after the object name. Essentially, as the display processing iterates over the specified object, the code chunk is applied to the specified values prior to the display. In Lisp, these can be expressed as functional closures and are pretty easy to manipulate. In C, we may have to provide a compiled library of code chunks that can be used in this way or with a parser.

Return Type none

Description see examples throughout this document

Exceptions

overlay-object

Arglist an object (this could be a virtual object)

Return Type

Description

Exceptions

Methods to set the Display Mapping Table

rgb-24, rgb-16, rgb-8

Arglist none

Return Type none

Description Specifies that the display of colors is mapped onto red green blue values on the display

Exceptions There are going to be several variants of this depending on how we want to specify allowable overlays values, deal with transparency. We should copy something simple from graphics. The different components can then be referred to as :red, :green, :blue.

overlay-color

Arglist Any from the enumerated set (red,green,blue,clear,....)

Return Type none

Description will perform the specified display in the overlay plane as a solid color.

Exceptions Assumes the display value is binary; if not, 0 gets mapped onto no display and other values get mapped onto display in the overlay color. The different overlay colors can be referred to by name or an associated number.

yiq

Arglist

Return Type

Description

Exceptions

hsu

Arglist

Return Type

Description

Exceptions

CLUT

Arglist

Return Type

Description

Exceptions

Transparency

Arglist

Return Type

Description

Exceptions

Flash

Arglist

Return Type

Description

Exceptions

Methods for Links between displays

create-link (link)

Arglist Interface-object1 Interface-object2 display-mapping-object

Return Type

Description

Exceptions

set-link

Arglist Interface-object1 Interface-object2 display-mapping-object

Return Type

Description Sets (changes) the display-mapping-object associated with the link between interface-objects

Exceptions

un-link

Arglist

Return Type

Description

Exceptions

Methods for Interaction

initial , always, continuous

Arglist Interface Command

Return Type

Description

Exceptions

interactive-function-selection

Arglist a set of Interface commands indexed by a number

Return Type

Description For an interactive function, programs are associated with different numbers. When a key is selected, the corresponding function is applied to values in the different lists of values obtained by mouse clicking.

Exceptions

clear

Arglist none

Return Type

Description Clears the position and value lists (probably need other for removing some number of items

Exceptions

4.4.3 Local Graphic Displays (vector, edges)

Local Graphic Displays are a subclass of object display, but instead of mapping an object attribute onto a screen intensity or color, it will display a parameterized graphic, such as a line, a square, a perspective view of a cube, or something from a library of such displays or a user specified one. A common example is a vector display which will map each component from a pair of images onto the x and y components of a vector, usually displayed as an overlay on top of an image. Other types of visualizations are possible. For visualizing three-dimensional attributes in register across an image, the user can display little unit cubes with their orientation computed from the specified components of display. The graphic display can be a piece of graphics code which will be positioned to the projected location of the pixel. For example:

```
[lg x-component y-component :graphic-type xy-vector :overcolor-plane red]
```

or the command language could include equivalent expressions such as

```
[v x-component y-component :overcolor-plane red]
```

will take two images and use them to specify the respective (x,y) components of vectors displayed in the red overlay plane at each point.

```
[lg x-component y-component
        :graphic-type xy-vector
        :x-increment 10
        :y-increment 10
        :value-function [+ x-component.value y-component.value]
        :linear-mapping 0 20 *min* *max*]
```

This displays a vector at every 10th pixel with the intensity of the displayed vector determined by summing the x and y components and then linearly mapping these between 0 and 20. Note the use of methods from the general display class.

```
[lg x-component y-component gradient-magnitude
        :graphic-type xy-vector
        :x-increment 10
        :y-increment 10
        :scale .5
        :value-function gradient-magnitude.value]
```

This displays the same vector field, but uses the third component to determine the intensity of the displayed vector. The scale method is expressed in object coordinates (accounting for the size of an object pixel relative to a screen pixel). There would also be methods for displaying vectors specified in (r,theta). (**Note**: we could have a method xy-vector-normalized.)

Edges are similar to vectors in drawing lines corresponding to the edges. Different types of edge displays should be distinguished:

- Mapping onto horizontal and vertical edges in the cracks between pixels
- Placing a single edge at the center of a pixel with it s orientation determined by the specified components objects

One way for displaying 3-dimensional surface orientation from images which contain the orientation angles:

```
[p image]
[lg angle1-component angle2-component
        :graphic-type unit-cube-perspective
        :overlay-color red
        :x-increment 10
        :y-increment 10]
```

4.4.4 Surfaces

SRI for terrain views of surfaces

Modification of existing package

4.4.5 Coordinate Transform Network of Images

There are several objects and displays which involve sets of related images, such as stereo pairs, motion sequences, pyramids, and mosaics. In some cases the relationship between the images is expressed by an explicit coordinate transform between the components of the object.

These displays can be handled in several ways:

- display each image in a different window and have them associated by links
- display each image in the same window but scale it to be appropriately registered with a selected scale and cycle through the display history
- display each image in the same window but choose a single scale for all of them and use general display mechanisms for indexing through all the displays.

4.5 Plotting Displays

Much of the functionality for plotting displays is defined in existing packages such as Mathematica. Mathematica has different types of plotting displays with "methods" corresponding to keyword-based argument passing. It would be nice to be able to use something like this.

There are some particular things that need to be addressed in using such a package that may require recoding into our environment:

- It should be compatible with the general display methods for such things as interacting with the plot display using a mouse; for the use of different colors; for being able to display.
- It should be fast and not require time-consuming conversion between the data formats of the plotting package and IUE objects.

Different types of displays include 1-dimensional plots, 2-dimensional plots, 3-dimensional renderings, scatter grams, density plots, parametric plots and contour plots.

4.6 Field Browsers

Browsers are generally used for interacting with the textual, symbolic and relational aspects of objects. Different browsers are used for different types of objects. There is one type for browsing a set or database of objects (set/database browser). There is another for inspecting attributes of a single object. There are two different types for viewing networks and graphs (graph browser and hierarchical browser). There is another type for inspecting spatial object which have coordinate systems or relations between coordinate systems (object registered browser).

(**Note**: Should browse methods be specialized based upon the type of object being browsed? Or how objects get mapped onto a particular browser? One type of interaction for dealing with lists, a structure, etc. How do we conform to the object inspectors in the development environment?)

Browsers have many similarities with displays. They can be linked. They have something like a value-function or display mapping object in that the characteristics of the browsers field (such as the text size, font, background color, display of an icon) are determined by the object being browsed.

(**Note**: How should links between display windows and browsers be handled? When a display is updated: the browser can be updated: when the browser is changed a value in the browser is changed, a display can be updated.)

Browsers are built from several component objects:

- A field appears as a rectangular box which has slots for

  - a background color
  - a text string
  - a text size, text font, text color
  - a type of boundary
  - a screen size and screen position
  - an icon
  - a field mapping object which describes how to map attributes of an object to the different of the field. For example, in the object registered browser, a field could contain a number colored as the intensity of the corresponding location in an image. Or the background of the field could be colored at this intensity and the text string shown in another color.

  A field also has a field-selection-action that is performed when the field is selected by a mouse-click. An action could be to bring up a browser on the object in the field that was selected. When a field is selected, what happens is much like when a mouse clicks a pixel in a display in the interactive mode. The location and value are stored in a list for later use.

- Fields can be organized into connected ***horizontal or vertical* field groups** where in each field as a unique index in the Field Group. The fields in field groups will generally have different object mapping functions. An example comes from the object-registered browser in which a given field group can correspond to registered values from different objects. For better visualization, these could be displayed in different colors, fonts, etc, in addition to their position in the field group. A field group can also have a distinct boundary.

- Field Groups can be organized into field matrices where in each group as a unique index set in the field matrix. This is used for mapping objects onto the matrix. Unless otherwise specified, the component fields of a Field Matrix will inherit the field attributes of the Field Matrix. There are constraints on the allowable sizes and positions of component fields.

- Field Matrices can be scrollable as a way to control the mapping of an object (or object set) onto the Field Matrix.

4.7 Object-Registered Browser

The object-registered browser is used to inspect the values in a neighborhood of a spatial object. A common example is inspecting the image values about a selected point. It is very much like the display of a spatial object in a display window, but instead of the values being mapped onto window positions and screen intensities and colors, things are usually mapped onto field locations and text. It is also possible to map onto general field values such as colored text, changes in font, colors, and icons.

The layout of an object-registered browser as it would appear for an image sequence is shown in Figure 4-5 . The fields display the specified values in a particular frame centered on the specified frame number and (x,y) position. In the actual display, the field which corresponds to the location of the array browser is shaded or highlighted. Spatial Directions are appropriately registered: up in the image means up in the array browser. The Object to Field Mapping Object specifies for a particular object what type of field mapping is used for it s specified attributes or values. This can be a user-specified function or from a library of existing functions. Users can select the type of font, text size, color, icons, that they want values in an object mapped onto.

For using the browsing an image and a segmentation:

```
[browse-object-registered
       image label-plane
       :field-action [browse *b3* [get-value label-plane
                                              object-location.x
                                              object-location.y]]
```

This indicates the importance of being able to access the object values and locations from the selected browser fields and the use of a object browser mapping object. This is very much like what occurs with mouse clicking in a display window for a displayed object: the values go into the object-value-list and the corresponding object locations into an object-location-list.

(**Note**: How would we perform array browsing on a pyramid? What is the local neighborhood? At given point, display the values in the children level. Assume there is an explicit coordinate transform between the different levels. The dimension button could then correspond to the things at different levels. What if the relation is a sequential one over time local neighborhood structure? Objects are related by some coordinate transform between the levels.)

(**Note**: How would we use a region to define the shape of the neighborhood? Use an object to perform a composition operation with another object? Use a curve to pull values out of an image? Use a containing volume, minimum bounding rectangle, minimum bounding cube and do regular array browse with respect to that?)

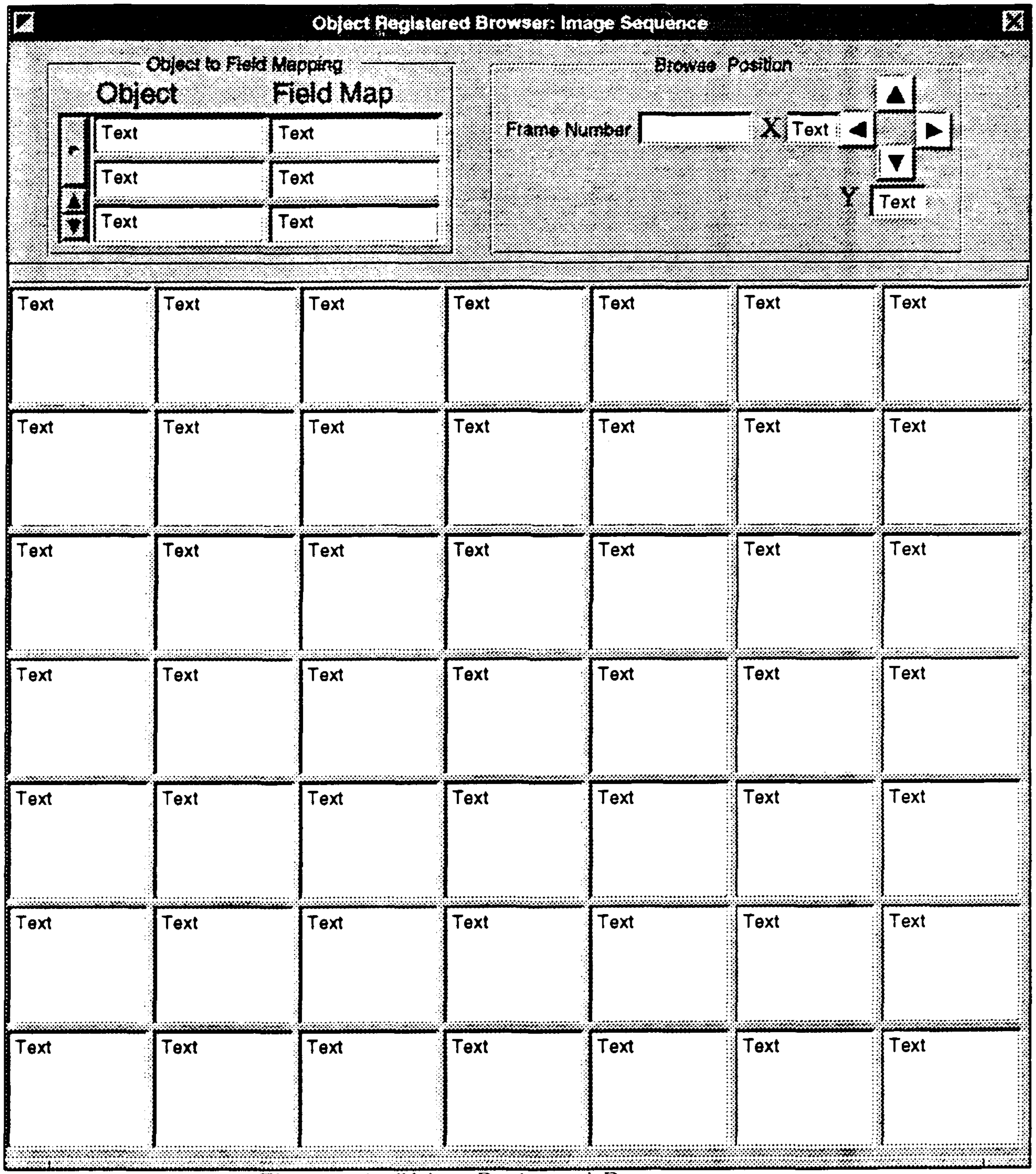


Figure 4-5: Object-Registered Browser.

4.8 Database/Set Browser

An example Database/Set Browser is shown in Figure 4-6. It consists of three basic parts: a field of button for interactively building queries, selecting objects, and applying operations. A text input field for typing in pieces of text to be used in queries. A field matrix in which rows will correspond to an object and columns to attributes the objects have in common.

For example, suppose I wanted to find a result that I generated but couldn't recall. In general, it would be a good idea to keep a browser over the set of active objects (let's call this the environmental Data Base - EDB). I would set this up with the command:

```
[create-db-browser EDB :attribute-fields name time-of-creation from-function]
```

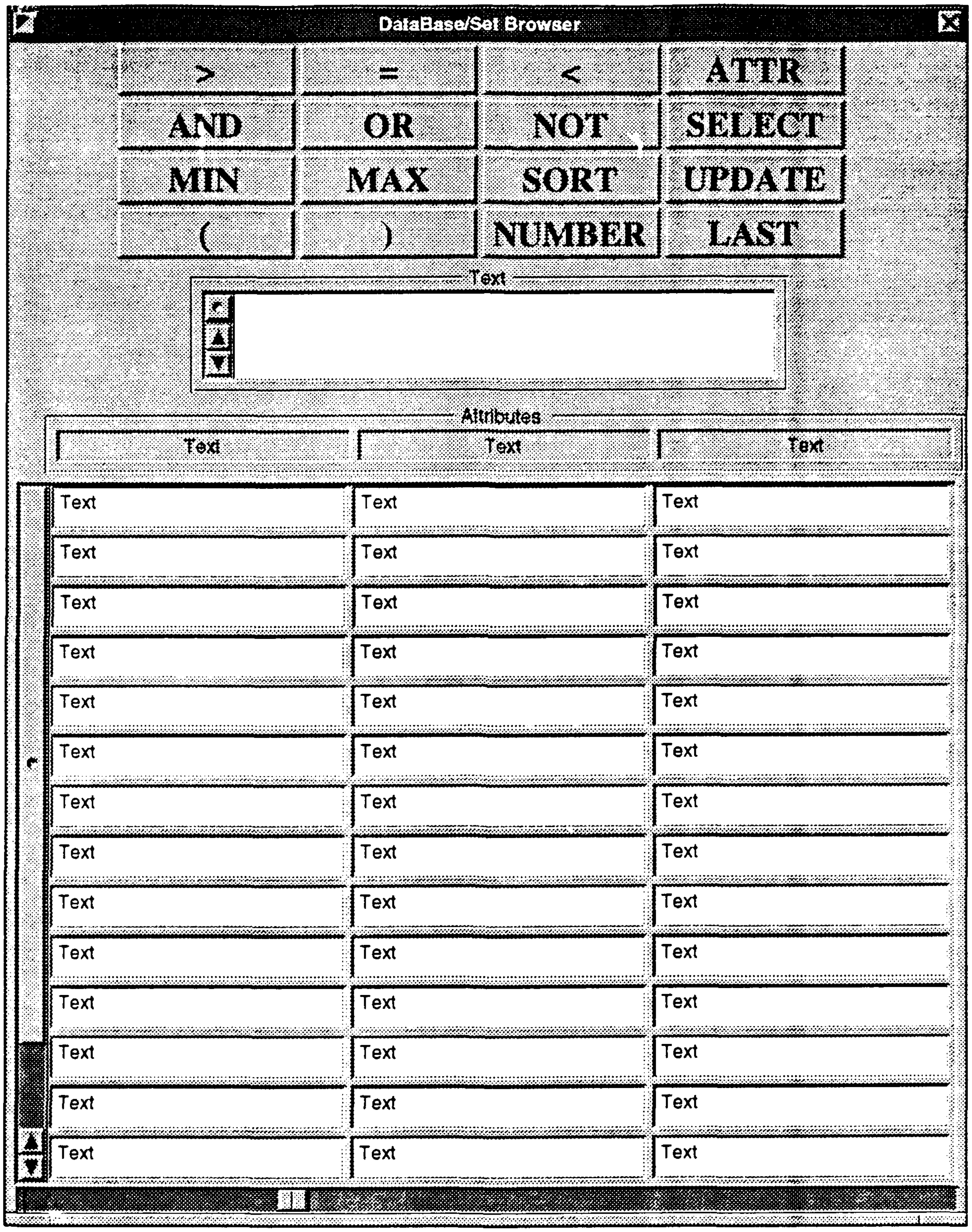
Sometime into my work, I want to find a flow-field that was the result of an amazing-function. I would type in the text area "amazing-function" [**Note**: for text-matching, substring matching could be used], click it, click =, click the column attribute field labeled "from-function", click **UPDATE**. Then the browser would list all the objects which had been created from the function *amazing function*. To find the most recent, I would click the attribute-field labeled "time-of-creation" and then click **GREATER-THAN**, **SORT**, **UPDATE**, and the topmost field would contain the most recently created object from this routine. This is also bound to the variable *current-object* and would be displayed as a default.

(**Note**: Perhaps, as a query is being created interactively, it should be displayed in an interactive command buffer. If so, how should this relate to the text input field in the browser?)

For each query, a set of selected objects is found. This set can be used for further queries. Sometime people make mistakes or want to return to a previously determined set. The **LAST** button is for this. We could save the current sequence of sets and continually back up through them by repeatedly clicking the **LAST** button (**Note**: This may be too costly).

It is also possible to associate with the text field selection action, the operation to browse the selected object in a separate object browser.

```
[browse EDB :selection-action [browse *b1* *selected-item*]]
```



Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text
Text	Text	Text

Figure 4-6: Set/Database Browser.

4.9 Object Browser

The object browser is a simplified version of the Database/Set Browser with out mechanisms for interactive queries. It is used for inspecting the attributes of a single object. One column is used for the names of attributes and the other column is used for their values.

4.10 Hierarchical Browser

The hierarchical browser is used for the inspection of graphical and network objects. Each column corresponds to a set of objects. When an object is selected, the types of relations (arcs) associated with the object are displayed in the *Current Arc Browser.* For a selected type of relation (arc), the related objects are then displayed in the next right column. This can then be repeated any number of times. It may be useful to have a field selection method to browse the currently selected object.

4.11 Graph Browsers

We need to define how to display IUE constraint networks. This includes: being able to zoom these in something like a display window, coloring the nodes based upon their attributes, and mapping them through a pixel *mapping function.*

Graph browsing a relational object or a graph will be set up using operations such as add-node, link-nodes, remove-nodes, add-arc, remove-arc, and interactive creation.

4.12 GUI Object Access

We need a general create-gizmo command.

It would be nice if the SRI model for menu definition for the environment could be set up in a browser and modified interactively.

4.13 Associated Objects

```
link-transformation object
parent display
child display
Object
Object-View mapping to be concatenated
```

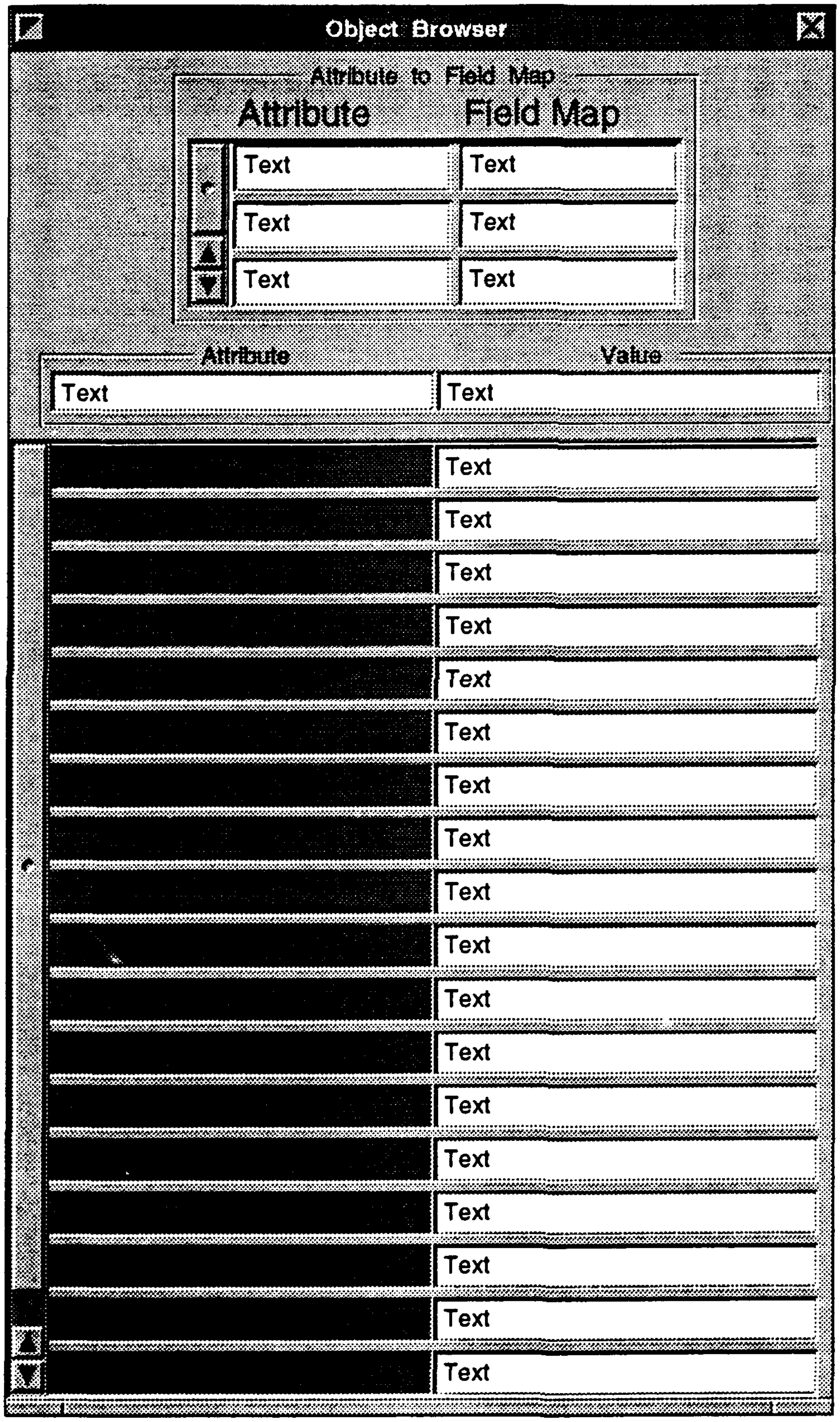


Figure 4-7: Object Browser.

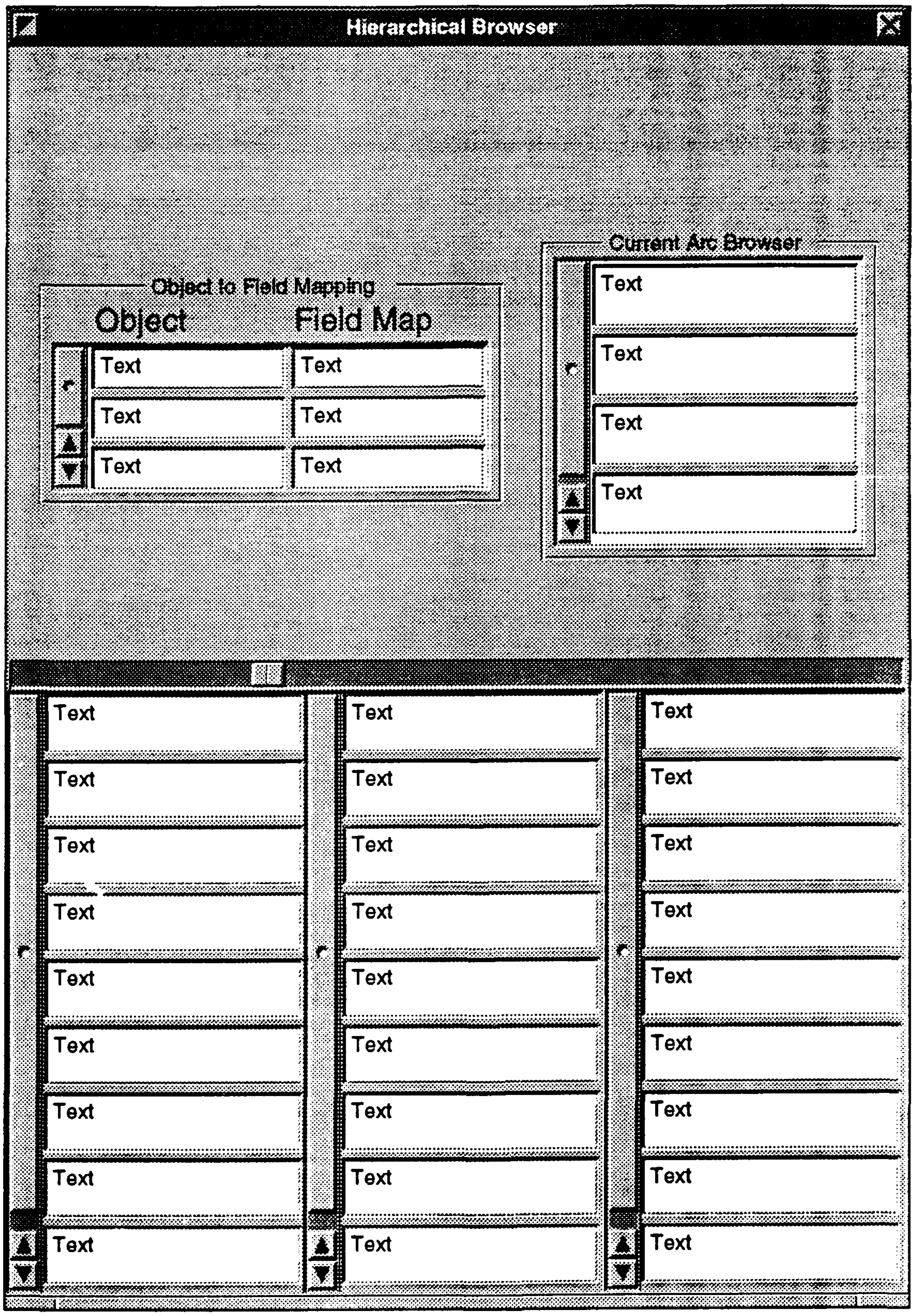


Figure 4-8: Hierarchical Browser.

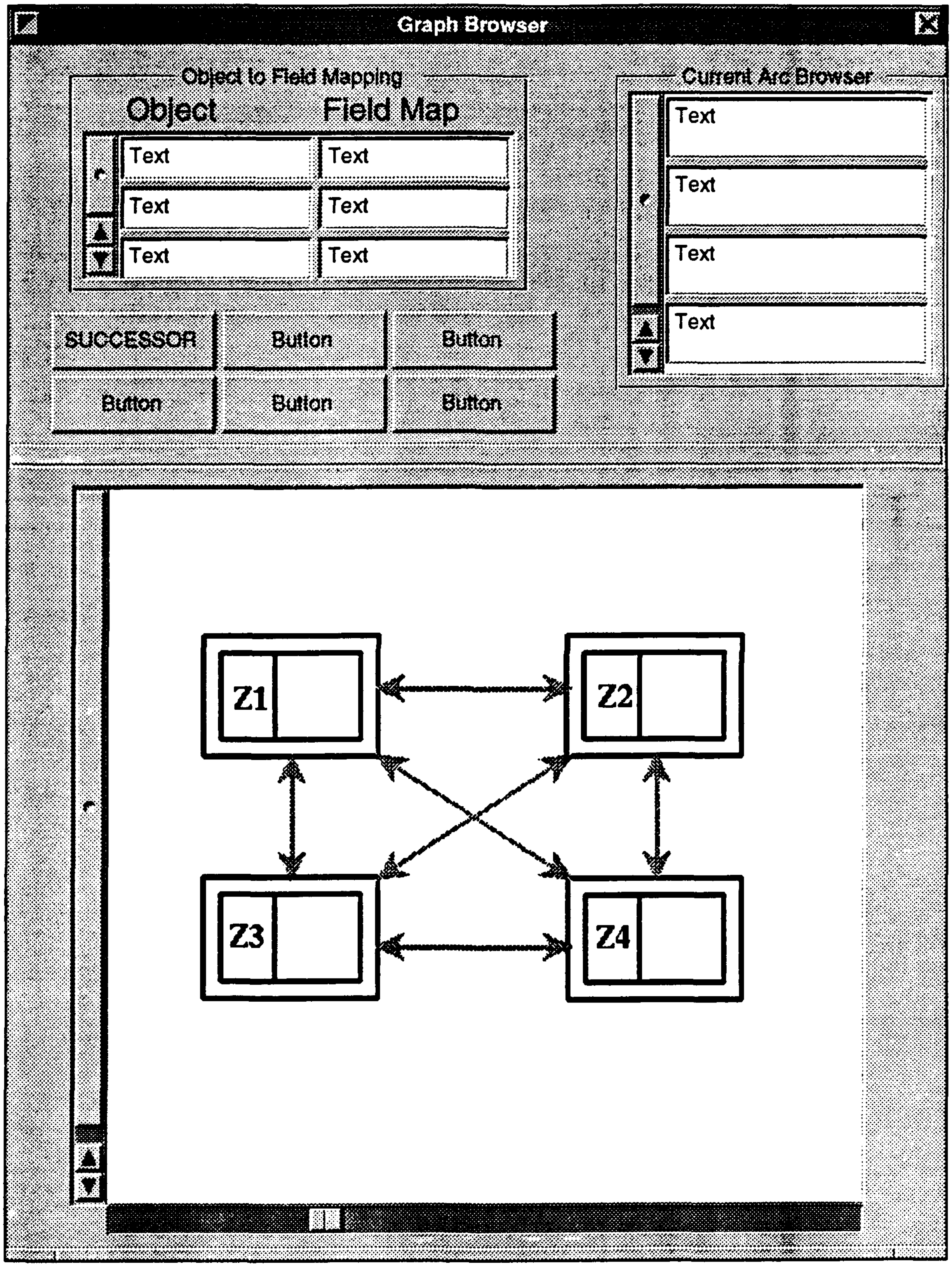


Figure 4-9: Graph Browser

```
Display Command Buffer Associated with window

Objects displayed in window

Object-View-Mapping
Object
Position Mapping
Value Mapping Table
Value Mapping Procedure
```

4.14 Interface Context

There are several values which are used to describe the current state of the interface.

- list of windows
- list of browsers
- links between windows and browsers
- current window
- current browser
- current display mapping
- current display mapping table
- the current object resulting from a browse or display action.

4.15 Interface Command Language

Messages can be sent to displays interactively through the command buffer.

All of the functionality of the interface is accessible through an interactive command language. The interface command language describes the overall functionality of the interface. All the interactive commands can also be placed in code for developing reusable scripts. The Command language supports defaults during interaction for conciseness and brevity.

4.15.1 Command Syntax

This is an initial stick in the ground. It comes from the display commands used in KBVision and PowerVision. The general syntax is

```
Display-action Window-Identifier object-set [keyword-arguments]*
                Snapshot File

Browse-action Browser-Identifier object-set [keyword-arguments]*
```

4.15.2 Window Creation

- Screen Size
- Screen Position

There are defaults maintained in global parameters describing the environment that are used if these are not specified. These can be changed from grabbing a window and resizing it in a manner consistent with the GUI package that is used.

`[cw` [1] `:size 512 512 :position 100 100]`

If a display operation is performed with no existing window, one is automatically created. Otherwise the *Current-Window* is used.

4.15.3 Browser Creation

4.15.4 Animation Commands

4.15.5 Window Linking

Display Windows can be Linked. A link between windows specifies a concatenation of functions to be applied to the domain and the range of the spatial object for mapping onto a display window. An example is using the display in one window to zoom onto a selected area in another window. In this case, the mapping from the domain of the object onto window 1 is concatenated with the mapping from the object onto window 2. Whenever a display mapping occurs in a window all of the associated linked windows are updates. For simplicity, displays are only updated when a display action is performed, not when changes are made to the spatial object. Cycles in window links are not allowed and therefore should be discouraged.

[1]See Section 4.15.9 on page 71 for a list of Interface Command Language Abbreviations

The basic commands for links are:

```
[create-link w1 w
    {followed by any of the keywords used to specify viewing an object}]

[unlink w1 w2]

[set-link w1 w2
    {followed by any of the keywordsused to specify viewing an object}]
```

For example,

```
[create-link w1 w2 :zoom 2 2 :pan 100 100]
```

creates a link between window w1 and w2 such that whatever is displayed in w1 will appear in w2, zoomed by a factor 2 and panned by (100,100).

```
[cw :rgb-24]
> *W1*
[cw :copy-attributes *W1*]
> *W2*
[cw :copy-attributes *W1*]
> *W3*
[cw :copy-attributes *W1*]
> *W4*
[link w1 w2 :value red-compcnent :red-8]
[link w1 w3 :value green-component :green-8]
[link w1 w4 :value blue-component :blue-8]
[p *W1* color-image]
```

Will display the different components of a color image in the three different windows.

(**Note**: Can windows can be linked to themselves (as in zooming into the same window)?)

Set-link changes the specified transformation between as also does a redisplay in the child window. For example,

```
[create-link w1 w2 :zoom 1 1]

[create-gizmo :type slider :range -5 5 :name zoom-slider
     :action [set-link w1 w2 :zoom zoom-slider.value zoom-slider.value]]
```

creates a link between two windows and a slider gizmo. When the slider gizmo is changed, it specifies how much zooming should occur and a redisplay occurs in w2 (the mapping from the spatial object onto w1 is not changed).

(**Note**: in interactive versions for zooming and panning should the selection square be an object, like a cursor?)

4.15.6 Object Position Mapping Keywords

- zoom
- pan
- affine mapping
- viewing coordinate transforms
- matrix
- position-function

4.15.7 Graphics

Often times the user will want to perform common graphical display for things such as text, simple two-dimensional graphics, more complicated three-dimensional graphics. Examples are annotating a display, indicating where some action is occurring (the position of an epipolar line, translational flow paths, etc.), projecting a wireframe of a model onto an image.

This functionality can be found in several existing graphics packages and it would be best if we could just link to such a package. An alternative is to come up with thousands of commands like

```
[draw-line 100 100 200 200 :color blue :thickness 3]
[draw-circle ....]
```

for circles, disks, text, and so forth, which will be specific to the IUE. This may be required if such graphics packages are not accessible in an interactive form for our interface.

It is important to distinguish three different modes in which graphic displays can take place:

- they can occur in the coordinate system of the display window. In this case displays only occur with respect to the window coordinate system.
- they can occur in the coordinate system of the displayed object. In this case I would be drawing a line with respect to the inverse mapping from window to object coordinates.
- when the display is performed with respect to the coordinate system of an object, it can actually generate an instance of an IUE object. Thus, in drawing a line in object coordinates, an instance of an IUE line object would actually be produced. When the wireframe model is displayed, each line-segment and junction will be created as an object in the IUE. For composite objects this may require significant processing. It is straight forward for simple objects such as polygons, curves, and so forth. This is very useful for producing data for testing routines. This mode can be coupled with the interactive processing mode to allow for the interaction creation of data.

These different modes could be specified by a global mode or as keywords in the commands.

4.15.8 Object Value Mapping Keywords

These are for specifying commonly used transformation from an object value onto a screen intensity or color:

- overlay-color. Can take values such as red, blue, green, clear (clear the overlay plane), cycle (cycle through a set of specified overlay colors so that features displayed at different times get different colors: will select the next available overlay color),user defined colors. If no value is specified, it means the display will take place in the overlay plane either using a default or as specified by a value-function. The overlay plane can be thought of as a glass sheet on which the user can write annotations and display extracted features and values. For a displayed image, the overlay will occur in register with the display window. For a displayed surface or volume, the overlay can occur with respect to the display window OR with respect to the surface or volume (**Note:** We need a way of specifying this).

- value-function: User can specify an arbitrary function which takes in an object value and returns a value to be displayed in a display window in terms of intensity or color. This is a little chunk of code that the display process uses when it decides how to color a value from the object. For example,

  ```
  [:overlay-color (red, green, blue, violet)]
  [p image :value-function [if (image.value > 10) red blue]]
  ```

  The first command tells the current display to change the display mapping table to use the specified overlay colors (in the display toolbox there will certainly be an interactive color editor for selecting and creating overlay colors). The second will display red in the overlay plane at a screen pixel corresponding to an image pixel if the image value is greater than 10, otherwise it will display blue.

  ```
  [p label-image :value-function [if (label-image.value = NULL)
                                   0
                                  (length label-image.value)]
                 :linear 0 20 0 *screen-max*]
  ```

  This function displays a label image (an image where each pixel contains a list of all the objects which occupy that pixel). The value function determines the number of objects in this list and the linear function maps this onto available screen intensities. The object value mappings are applied in the order of the keywords.

- **overlay-object** the object which will be overlaid. This is generally used when one object determined where something will be displayed and another determines what color it should be. For example, suppose I have a binary edge image that I want to overlay on top of an color image:

  ```
  [p image]
  [p edge-image :overlay-color red]
  ```

  this could be specified as:

```
[p image :overlay-object edge-image :overlay-color red]
```

A typical use of this is for displaying extracted features with respect to a surface. Suppose I have a surface display of the intensity values in an image and I want to display the locations of an extracted edge-image on top of this:

```
[s image :overlay-object edge-image :overlay-color red]
```

- CLUT: User specifies a color look up table to map display object values through. This can be an array of numbers or an array of functions (pointers to functions).
- transparency operators: (**Note**: Consider implementations on NeXT and Silicon Graphics)
- linear: User specifies a range of values in the object and the range of screen value that they will be mapped on to define a linear function.

  ```
  [p image :linear 0.0 256.0 screen-min screen-max]
  ```

  would display image in the current window and linearly map object values from 0 to 256 onto the range specified by the system globals screen-min and screen-max (these would probably be defaults)

  ```
  [p image :linear 0.0 256.0 CLUT[1] CLUT[2]]
  ```

  map between values found in the specified color look-up table

4.15.9 Interface Command Language Abbreviations

p for pixel

i for interactive

v for vector

b for browse

cw for create window

cb for create browser

g for graphics

s for surface

4.15.10 Interface Context and Intelligent Defaulting

4.15.11 Organization of System Menus

4.16 Implementation of Nice Features/Examples

This section shows how nice features found in other IUE system will be implemented in the IUEUI by using its basic objects. It also presents examples for typical or interesting interface operations.

4.16.1 Displaying an Image

The simplest way of doing this is to hit the display button on a display window or to type the command

```
[p]
```

and the *current object* will be displayed in the *current window* using the *current object display mapping* and the links to other windows and browsers associated with the *current window*.

A different current window can be selected by clicking anywhere on a window. The *current object* can be selected interactively from a browser. For example the command

```
[browse Objects-from-session.September20]
```

would either create or use the *Current-Browser* on all the objects saved in a database from a session on September20. The user would then use the command buttons associated with the browser to select an object as the *current object* and then type

```
[p]
```

The selected object will be displayed using a default *object display-mapping*. If no window exists, one will automatically be created based upon the object and system defaults.

The user can play with the *object display mapping* using keywords. For example,

```
[p :CLUT clut51]
```

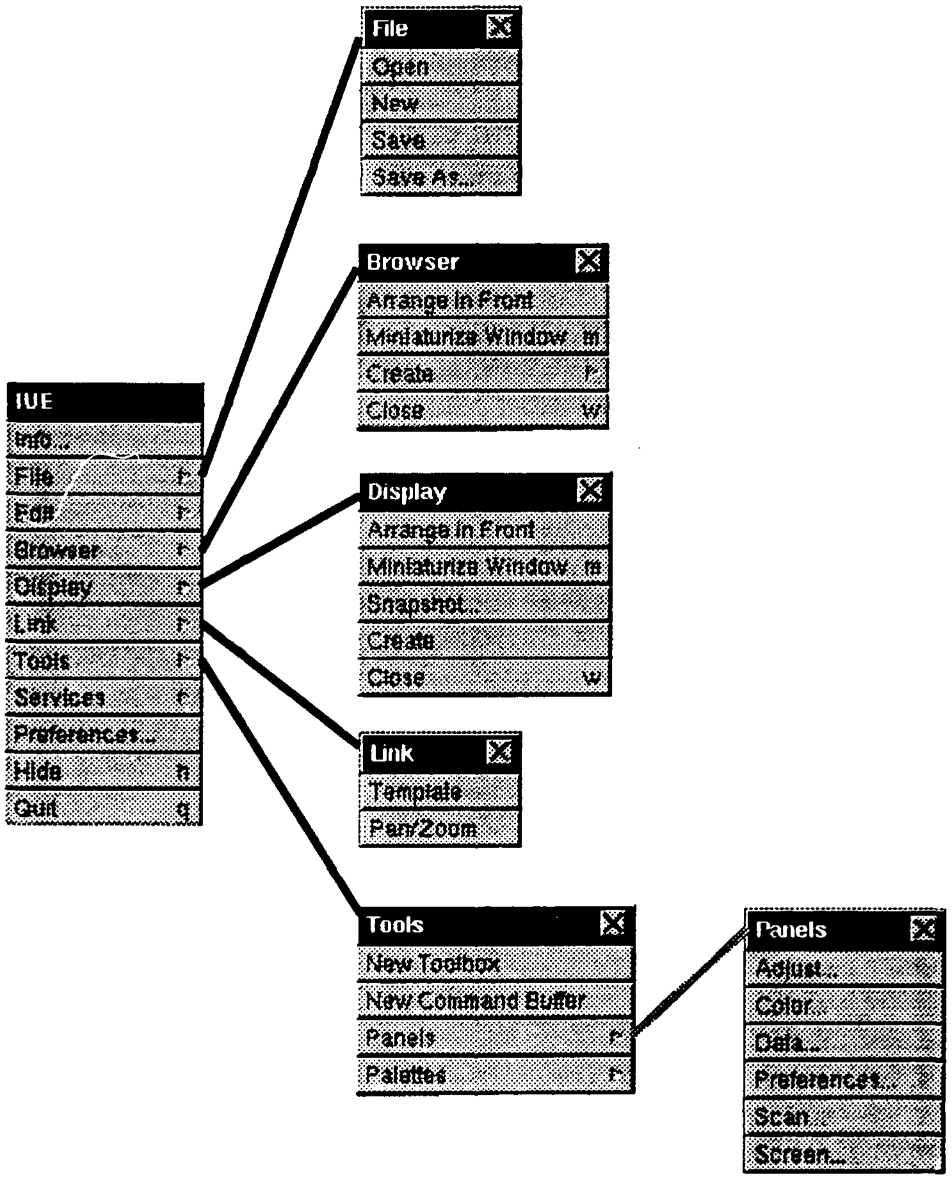


Figure 4-10: System Menu Layout

would display using the current defaults while the user supplies the color look-up table to map from object values to screen values.

```
[p :linear 0 128 *screen-min* *screen-max*]
```

would display using the current defaults while the range of object values from 0 to 128 are linearly mapped onto the range of values *screen-min* and *screen-max* (these could be defaults).

Additionally the user could type

```
[create-gizmo :type slider
              :name slick
              :min 0 :max 256
              :action
      [p image :linear 0 [read slick] *screen-min* *screen-max*]]
```

4.16.2 Interactive Function Application

The interactive display allows a user to access the currently displayed object or a list of specified objects through a display window using a mouse, pointing device or other interactive device (space ball, data glove). We begin with the simple form of this where we assume the user has a mouse and a keyboard. The command [This may belong as a menu-level command or a default when clicking in the window]

```
[i]
```

places the user in interactive mode with respect the *current window* and the *current object*. Thereafter, when the user clicks in locations in the window, he is returned the corresponding locations and values in the displayed spatial object. These values can be displayed in the Command Interface or in the small text field associated with the current window. What is actually happening each time a click occurs in the window is that the queues for *object-positions*, *object-values*, *window-positions*, *window-values* are being filled with the selected values.

```
[i image1 image2 image3]
```

will do this for all the specified images simultaneously and store lists of values in the queues. To exit interactive mode, the user will hit some specified mouse button or keyboard key. (**Note**: What about viewing objects such as closed surfaces which may be two-sided? We assume that the IUE will provide basic operation for intersecting a ray of projection with a set of specified objects with respect to a coordinate system. This operation could be invoked from the values returned by the i command. In general, all the geometric operations should be general ones to the IUE which can be invoked through the interface.)

The user can also call functions in the interactive mode to be applied to the values in the different queues. For example:

```
[i image  :1 [p :overlay-plane clear]
             [p image :value-function '(if (image.value > *object-value*[1])
                                            red blue)]
```

Whenever the user hits the terminal key 1, the overlay plane will be cleared and all image locations with a value greater than the value at the selected image location will be displayed in red, otherwise blue, in the overlay planes. *image.value* is a dummy variable that refers to the current value in image which is being displayed. **object-value*[1]* refers to the value selected using a mouse click in the display window. *red blue* refers to globally defined overlay colors. Recall that the *:value function* specifies the operation to be applied to an object value to map it onto a screen intensity or color(**Note**: should there be a default for transparency operations?). Values in image are returned whenever the user clicks the mouse.

Another example:

```
[p *W1* image1]
[cw :copy-attributes *W2*]
[p *W2* image2]
[i *W1* image1 :1 [g *W2* :draw-line
                              *W1.min-x*
                              *object location*[1].y
                              *W2.max-x*
                              *object-location*[1].y]]
```

The user displays image1 in *W1*. He then creates another window which is identical to *W2* and interactively moves it to another screen position. He displays image2 in this new window using the *current-object-display-mapping*. Thereafter, when he interactively selects a point in image1, the corresponding epipolar line (no rotation, only translation in X) is displayed with respect to image2.

Let's say the user has a label image containing a set of extracted image features and he want to inspect these in detail.

```
[cw :size 512 512 :name zoomie]
> *W1*
[cw :size 128 128 :name zoomer]
> *W2*
[create-link *W1* *W2* :transformation-link :zoom 5 5 :pan 0 0]
[cb :type object :field-number 5]
[p *W1* image]
[p curveDB :positions curveDB.locations :overlay red]

[i label-image :always [browse *object-value*]
                       [set-link *W1* *W2* :pan *object-position*[1].x
                                                *object-position*[1].y]]
```

Here the user creates two windows and displays the image and the extracted database of curves in one of them. This is then displayed in *W2* under the specified transformation. The user creates an object browser. When the user clicks in *W1* with respect to the label plane, the curve object at that location is returned as the *object-value*. This is then browsed in the current browser and the area surrounding this curve is displayed in *W2* to get a close-up on the curve and the underlying image.

```
[i image
   :initial [message "select endpoints and press 1"]
   :1       line-mask = [make-instance
                                :object line
                                :domain-increment 1
                                :interpolation nearest-neighbor
                                :pt1 *object-position*[2]
                                :pt2 *object-position*[1]]
   [p :overlay clear]
```

```
[p line-mask :positions line-mask.locations :overlay-plane red]
[plot1d *W2* :y-values [object-compose line-mask.positions image]]
```

Here the user is interactively creating a line-mask with mouse action, displaying the line-mask with respect to the image, and then plotting the image values found along the line-mask in as a 1d plot. Many things in this example are operations in the IUE that are still being specified. `line-mask = [make-instance :object line` creates an instance of a line object with integer indices that are mapped onto the image coordinates between pt1 and pt2 using nearest-neighbor interpolation. Such a functionality will exist in the IUE. There is also the issue of how we handle local variables (such as line- mask) in the IUEUI command buffer. Once the line-mask is created, it is displayed in the overlay plane. Finally, the composition of the positions in the line line-mask with the image (basically composing the functions line-mask: integers $-/>$ 2D Image Coordinates and image: 2D Image Coordinates $->$ scalar values) are plotted.

4.16.3 Displaying a set of Region Boundaries and Junctions

The user would type

```
[p image]
```

to display the image from which the features were extracted. RegionDB and JunctionDB refer to a database of extracted regions and junctions. To display these the user could type

```
[p RegionDB.locations :overlay cycle]
[p JunctionDB.locations :overlay green]
```

This would display the regions by cycling through the different defined overlay colors and then displaying the junctions in the green overlay plane. There are problems with this: adjacent regions may have the same overlay color and green junctions may be overlaid on top of green regions. The adjacent region problem could require a region coloring option (or the user could display the region contours in an overlay plane). The other problem could be solved by using compositing options such as exclusive-oring or by commands such as

```
[p RegionDB.locations :overlay cycle (red,blue,yellow,pink,orange) ]
[p JunctionDB.locations :overlay green]
```

where the set of overlay values to cycle through is explicitly listed (and doesn't include green). (**Note**: Another possibility is to animate the overlays so they flash using double buffering animation).

RegionDB.locations reflects some general issue with how to refer to IUE objects and their attributes during display operations. First, we are referring not to a single object but to a set of objects that we want displayed. The display method will have to iterate over the elements of this set applying the appropriate display method to each. Second, regions and junctions have several attributes and we need to be clear about exactly what we want displayed from them. Here we are referring to the locations in image coordinates that are occupied by these objects. Let's say we wanted to color the regions based upon some attribute such as the value of some associated texture measure (what will these references look like in the IUE?). We suggest something like

```
[p RegionDB :positions RegionDB.locations :values
RegionDB.texture-measure
            :linear 0 100 *min* *max* :red-8]
```

This says to display the RegionDB with the positions coming from the locations attribute of the regions in the regionsDB and the values by taking the Region DB texture mappings and using a linear mapping from these onto screen intensities in 8 bits of red.

or

```
[p RegionDB :positions locations :values texture-measure
            :linear-mapping 0 100 *min* *max* :red-8]
```

Where it is locations and texture-measure are defined attributes of region objects in the regionDB.

4.16.4 Display of a Vector Field

```
[p image]
[v image.gradient :x-inc 5 :y-inc 5 :length-scale 5]
```

each component is being used to specify the components of the displayed vector. This should be a general operation where the components are used to describe the display of an arbitrary graphical object, such as a perspective display of a cube to provide information about orientation

4.16.5 Window to window Zooming and Panning

Creation of windows and browsers and creating links between these is a basic processing mode and can be performed through system menus. The user would select create-window from the system menu and create two windows by clicking and dragging the mouse to size the windows and would then position the windows by dragging the menu-bars. He could select [either off a menu or from a button in the title bar of the window] the interactive window tools/inspector to set attributes such as name, title bar color.He would then select link-windows from the system menu. The order in which he then clicks on the windows creates a link between them. A browser of transformation between windows comes up that he can select from. He would select zoom and pan and specify it using a dynamic selection rectangle. Thereafter, whenever a display occurs in window *W1* the same display occurs in window *W2* with the transformation specified by the link.

Alternatively, this could all be done through interface commands:

```
[cw :screen-position 100 100 :screen-size 256 256 :name "big" :bar-color red
> *W1*
[cw :screen-position 100 300 :screen-size 256 256 :name "zoomer"]
> *W2*
[link *W1* *W2* :zoom 2 2 :pan 100 100]

[i  :1 [set-link *W1* *W2* :zoom {add (1 1) to current zoom value}]
    :2 [set-link *W1* *W2* :zoom {subtract (1 1) to current zoom value}]
    :3 [set-link *W1* *W2* :pan object.location[1].x object.location[1].y]]
```

4.16.6 Browsing a selected Image Area

The simplest way to do this would be to type

```
        [array-browse]
```

and an array browser will be set up on the *current object* at a default location and neighborhood size (probably the center and 10) using the *current object browse mapping* [what type of font is used, etc.] The dimension of the array is based upon the dimension of the object. The user could use the command buttons on the array browser to move the inspection point through the object (1D or 2D arrow pairs plus another for selecting dimension—the default setting would come from the browse method associated with the object to be browsed). Or the user could type

```
        [array-browse object]
```

if the user types

```
        [array-browse object1 object2 ...]
```

the fields in the array browsers will display registered values from all the specified objects. If the user types

```
        [array-browse object1 [Function1 object1] [Function2 object1]]
```

the fields in the array browsers will display registered values from object1 and the application of Functions 1 and 2 to object 1. If I have two different routines for computing curvature along a curve and I want to inspect them, I could type

```
        [array-browse curve [curvature1 curve] [curvature2 curve]]
```

An alternative is to run the array browser on mouse selected locations. This may be a default action relative to the current window or the user may type

```
        [i :continuous [array-browse
                          :location *current- object.x* *current-object.y*]]
```

and the browser will display values in a neighborhood surrounding the current object position selected by the mouse. The neighborhood will be updated continuously as the mouse is moved.

Setting up links between array browsers: the links have associated processing actions to see the effect of a routine over a selected neighborhood.

4.16.7 Displaying a Color Image

```
[p color-image :color rgb-8]
```

displays the color image in with 8 bits in red, green and blue.

```
[p image1 image2 image3 :color rgb-8]
```

treats the three images as specifying the rgb components

4.16.8 Multiple Image Display

```
[p image1 image2 :values [- image1.val image2.val]
        :linear-range -20 20 *min* *max*]
```

4.16.9 Interactive Inspection of Image Registered Features

This involves how spatial queries are performed in the IUE. One way involves the use of a label image. This is essentially a depth buffer which stores a list of objects which occupy a given pixel (the list could be ordered by depth or time of extraction of feature). This can be treated as an image and all the operations that can be applied to an image can be applied to it. Such as

```
[p label-image :values [if (label-image.value  /= NULL)
[i label-image :1 [browse label-image.value]]
```

4.16.10 Finding a particular result and Displaying it

The user would browse a particular database by typing

```
[browse long-term-data-base
        :attributes (time-of-creation,function,type)]
```

and a browser would come up with respect to the objects in the long-term data base listed row by row with the attributes time-of-creation, function, type in the columns. The user would click the column head **type** and then = and then type in "image" and the browser would then display all the objects of type "image". The user would then click the column head **function** and the = and then type in "Gaussian" and the browser would then display all the images created by the Gaussian Routine. The user would then click the column head Time-of-creation and then Sort Max and the browser displays the selected images ordered by their time of creation. By clicking to the side of the first listed image, it then becomes the *current-object* and can be displayed either by clicking the display button in the browser or by typing

```
        [p]
```

Suppose the user had a database of regions and wanted to find the largest. He would type

```
        [browse RegionDB :attributes area ]
```

4.16.11 Animation of Surface Display of a moving edge

Here we are trying to understand the nature of an edge in a sequence of images. To do this, we display the intensity in a selected image area as a surface plot and then display, as an overlay, on top of the surface plot, the extracted edges. This is done using a sequence of images and extracted edge-images (binary images where 1 indicates the presence of an edge and 0 the absence) and writing the display out to an animation file and then playing it back in a window.

To do this, the user would type:

```
{iterate image over image-sequence and
         edge-image over edge-image-sequence

        [s animation-file image :image-position 200 200
                                :distance 100
                                :Theta1 20
                                :Theta2 3
            :overlay-object edge-image ·overlay-plane red :clear t]}

[play animation-file]
```

Several things are occurring here. The first two lines correspond to the set iteration control structures provided in the base programming language. We don't want to build a complete programming language into the interface command language so it's not clear if this is interactive. These lines cause the variables image and edge-image to iterate over the respective sets. The next line says to do a surface display to an animation file. Instead of displaying in a window directly, the display will be written out to a file for play back. If a window were specified instead, the display would occur in a window (although this may not play at an effective speed for an animation). The surface display selects an image point location which will be viewed, viewing distance from this point and the angles of view. It also states that any previous display actions will be cleared (it's a new frame) and that the edge-image is displayed in red values at locations it occupies on top of the intensity surface.

The play command will play the animation back in the current window. There are keywords associated with the play command for controlling the speed of the animation. In fact, these parameters could be linked to a slider for interactive control.

Animations can be made on the fly. A user can always specify this by indicating an animation file instead of a display window in a display command. Another command is

```
[snapshot window animation-file]
```

which will store whatever the state of the display in the specified window is out to an animation file.

4.16.12 Interactive Surface Display

Here the user specifies a location in an image in one window and in another window, a surface plot around this location is displayed. The user would type:

```
[p w1 image]
[i w1 :1 [s w2 image :location image.x image.y
                     :distance 200
                     :theta1 30
                     :theta2
                     :clear t]]
```

another possibility would be

```
[create-gizmo :type slider :range 0 1000 :name distance]
[create-gizmo :type slider :range 0 180 :name theta1]
[create-gizmo :type slider :range 0 180 :name theta2]
[p w1 image]
[i w1 :1 [s w2 image :location image.x image.y
```

```
                                        :distance slider.distance
                                        :theta1 slider.theta1
                                        :theta2 slider.theta2
                                        :clear t]]
```

another possibility would be to allow variables that could be set by expressions in the interface.

4.16.13 Process Monitoring

This is essentially a Set/DataBase Browser on a database of task objects. To be KBV-ish, one of the field column should be mapped onto red or green. Tasks would change the colors in their corresponding fields based upon their status.

4.16.14 Constraint Network Monitoring

This section needs to be created in coordination with the design of the IUE constraint networks.

4.16.15 Interactive Histogram Segmentation

Clicking on (or near the axis of a plotted function) returns the x coordinate and the y-value of the displayed object.

```
        [plot2d *W1* histogram]
        [p *W2* image]
        [i *W1* histogram :1 [min = current-value[1].x]
                             [max = current-value[2].x]
                             [p *W2* image
                                :value-function
                 [if ((image.value > min) & (image.value < max)) blue]]]
```

Here the user has plotted a histogram in *W1*. He then selects the range of values by clicking on the displayed histogram. The current-object-value contains the x and y value from the displayed histogram. These are stored in the local values min and max (this isn't necessary: in general how will we refer to local variables in these expressions? Will there be a set of dummy variables?).When the user hits the key 2, the selected range of values are displayed in the blue overlay plane.

Section 5

Summary

The IUE will be a revolutionary system covering many technologies, serving many levels of users, and growing by distributed development over many years. The class hierarchy framework and user interface concepts presented in this document will allow:

- Integration of the diverse concepts of IU within one environment.
- Rapid introduction of new users to the IUE.
- Organized extension of the base IUE by developers (including new users).